All I have done is give you a book.
You have to have the courage to learn what's inside it.

Miss F. J. Riley from Homer Hickam's Rocket Boys: A memoir

University of Alberta

INTEGRATING COMPUTATIONAL RAM (C•RAM) INTO A SYSTEM ARCHITECTURE

by

Noah Aklilu ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Electrical and Computer Engineering

Edmonton, Alberta
Fall 2001

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Integrating Computational RAM (C•RAM) into A System Architecture** submitted by Noah Aklilu in partial fulfillment of the requirements for the degree of **Master of Science**.

To my parents, who moved to a new land for their children.
To my sisters, who were a source of advice and support.
To Tari, who became my cheerleader.

Abstract

This work represents research that was conducted in integrating Computational RAM (C•RAM), a processor-in-memory technology, into a system architecture. This integration would provide an economical parallel processing solution for data-parallel computation in areas such as signal processing, fault simulation, and data mining. To facilitate this integration, a controller architecture is presented, which provides a host processor with low latency access to the memory functions and SIMD capabilities of a C•RAM-based SIMD array. A hardware platform, the Embedded SIMD Machine (ESM), implements a controller designed to function with a C•RAM device equipped with 4096 PEs and 13.4 Mbits of DRAM memory. Algebraic kernels are used to compare the performance of the ESM to other systems. Based on the performance of the ESM, the impact of replacing a system's main DRAM memory with C•RAM is analyzed and the results are presented, showing performance improvements of over an order of magnitude for a system with 128K processors.

Acknowledgements

This work was made possible through the support of many people and organizations. Within the department of Electrical and Computer Engineering the author wishes to thank: Dr. Duncan Elliott, my supervisor, for providing guidance, funding, and logistics support; Dr. Bruce Cockburn for providing a second opinion and sitting on my supervisory committee; and the office staff, including Michelle Lock, who helped in understanding the system of graduate school. Outside of the department the author wishes to thank: the people of Mosaid Technologies and Accelerix for technical support, especially Randy Torrance and David Jones (formerly of Mosaid); Mark Geddes of Windriver Systems for his valuable feedback during the design of the second C•RAM daughter-board; Marco Castellon for allowing me to use his contacts for information; Kaytronics for donating the NEC MIPS evaluation board; Curtis Wickman for designing the first C•RAM daughter-board; and Tari Hiebert for editing the manuscript of this thesis.

The author would like to thank the following individuals for granting permission to use their figures:
D. Elliott for Figures 2.2, 2.3, 2.4, 2.5, and 2.6
P. Nyasulu for Figure 2.12.

Contents

List of Figures

List of Tables

List of Symbols

ALU	Arithmetic Logic Unit
C•RAM	Computational Random Access Memory
DRAM	Dynamic Random Access Memory
EAB	Embedded Array Block
FIFO	First In First Out
FPGA	Field Programmable Gate Array
LE	Logic Enhanced
MIMD	Multiple Instruction streams Multiple Data streams
PE	Processing Element
PIM	Processors In Memory
RAM	Random Access Memory
ROM	Read Only Memory
SIMD	Single Instruction stream Multiple Data streams
SRAM	Static Random Access Memory
VRAM	Video Random Access Memory

Chapter 1

Introduction

1.1 Overview

This thesis presents research conducted in integrating a parallel processor architecture into a microprocessor-based system. The result is an economical parallel processing platform that is capable of tackling particular tasks with significant improvements in performance. The parallel architecture used to achieve this is computational RAM (C•RAM) [1], a technology that integrates simple processors with a memory core inside a single device.

Today's microprocessors face a hurdle that is the result of their phenomenal improvements in performance. The bandwidth available to access memory has not been able to keep up with these improvements. The bandwidth available through the external bus of a memory device is a fraction of what the microprocessor requires and a fraction of that available internally within the device. Depending on the memory device, the microprocessor might have access to less than 1% of the bandwidth available internal to the memory device. This problem is referred to as the "Memory Wall" [2]. In a traditional system the microprocessor has access only to the reduced bandwidth offered by the external bus of the memory device. By implementing processors inside a memory chip, C•RAM makes it possible to move some of the computational work inside the memory chip to tap the internal bandwidth. Based on a Single Instruction stream Multiple Data streams (SIMD) model of parallel processing, C•RAM uses simple processors to create a massively parallel architecture within the memory device. The processors, which are bit-serial, are intended to be implemented in large numbers from thousands to tens of thousands. The largest implementation to date for a single device is 4096 processors in the Accelerix AX256 graphics accelerator [3].

The SIMD architecture of C•RAM and its simple processors might seem to restrict its capabilities, but it has demonstrated performance improvements in a number of areas including graphics processing, data mining and fault simulation [4] [5]. C•RAM can be used to improve the performance of a system for applications where parallelism exists in the data.

To integrate C•RAM into a system architecture, a C•RAM controller is required to coordinate access to the C•RAM device. C•RAM requires a controller that coordinates memory access as well as the issue of SIMD instructions, because of C•RAM's dual role as a parallel processor and a memory device. We present a C•RAM controller which is later implemented in an embedded system, the Embedded SIMD Machine (ESM). While SIMD parallel processing has a heritage of machines and their associated controllers, this C•RAM controller deviates from the methodology used by most of the past SIMD machines. The controller depends on the host, in this case a microprocessor, to perform some of the work related to issuing SIMD instructions to the C•RAM device. This simplifies the C•RAM controller and reduces the latency, which it introduces in the communication path between the microprocessor and the C•RAM device. Instruction buffering and the addition of a microcoded sequencer are discussed as possible enhancements to the proposed controller. For applications that overlap sequential and SIMD instructions, these enhancements are intended to reduce the workload of the host in issuing SIMD instructions. The inclusion of these enhancements, however, increases the complexity of the controller.

The implementation of a system, the Embedded SIMD Machine (ESM), that uses our proposed controller is described. The ESM is composed of a Mitsubishi MSA2000 microprocessor evaluation board with a custom designed daughter-board. The M32R/D microprocessor on the MSA2000 board is the system host, and the C•RAM controller and a AX256 device with 4096 processors are on the attached daughter-board. The controller is implemented in VHDL and the synthesized circuit is programmed into an Altera 10K100A FPGA. In this FPGA the controller requires 393 logic cells out of the 4992 cells available.

In the absence of a functional AX256 device, the ESM can take advantage of the deterministic timing of the AX256 to emulate its presence for the purposes of evaluating system performance. This hardware emulation is used to evaluate the ESM and compare its performance against a control group of systems. Using three data-parallel kernels[1] that represent operations in digital signal processing, the ESM is compared to a Sun Ultra 10 workstation and an MSA2000 embedded system (essentially the ESM without C•RAM). The results show the ESM performing better in one kernel than the Ultra 10, but performing worse in the other two. When compared to the MSA2000 without a C•RAM attachment, the ESM performs better on all kernels with speedups that range from 3x to 75x. During the execution of the kernels, the ESM achieved a high PE utilization of over 50% because of its controller architecture. The improvements realized by the ESM over the MSA2000 present C•RAM as a possible candidate for implementing features such as multimedia in an embedded system. Depending on the application, this would require a digital signal processor in addition to the microprocessor, increasing the power budget and cost for the system. When compared to ordinary memory, a C•RAM implementation adds an increase of 3% to 20% in area and a power overhead of 10% to 25% to a memory core [5]. This presents C•RAM as a cost-effective and power-efficient alternative for implementing signal processing, which can be used as system memory or when required as a powerful SIMD parallel processor.

The results of the comparison between the ESM and the Ultra 10 workstation demonstrate that a higher level of parallelism is required to gain a significant benefit from C•RAM in a workstation environment. Using the performance of the ESM as a measure, we analyze the impact of C•RAM in a workstation equipped with 4K processors to 128K processors. With a 128K processors, the results from the data-parallel kernels show improvements of over an order of magnitude. For scientific applications that can utilize a large number of SIMD parallel processors, such as neural networks [6], C•RAM can be a worthwhile addition to a workstation.

[1]We use the term kernel to signify a small mathematical benchmark

Based on our research, we present an architecture for a low latency C•RAM controller that integrates a C•RAM device into a system. We demonstrate that C•RAM is capable of improving the performance of a system, by implementing a controller based on our proposed architecture and benchmarking the system. In a system C•RAM can be used as traditional memory and, as required, a parallel processing architecture. A C•RAM equipped system can realize significant improvements in performance for a wide range of applications that can benefit from a SIMD parallel processor, such as signal processing and fault simulation.

1.2 Thesis Organization

This thesis is organized into six chapters. Chapter 2 presents background material. Chapter 3 presents our C•RAM controller and the methodology behind it. Possible enhancements to the controller are also discussed. Chapter 4 describes the Embedded SIMD Machine, which is the hardware platform that implements our C•RAM controller. In Chapter 5 the ESM is evaluated and compared to systems acting as controls. Additional analysis on C•RAM performance is performed using the results of the ESM evaluation. Chapter 6 summarizes and concludes the thesis.

Chapter 2

Background

This chapter is presented in three sections: SIMD architectures, the C•RAM architecture, and controllers for SIMD devices including C•RAM devices.

Section 2.1, SIMD architectures, introduces SIMD as a parallel processing architecture. A comparison with MIMD highlights the differences between these two parallel architectures and the advantages of SIMD.

Section 2.2, C•RAM, discusses the development of computational RAM (C•RAM) as an integration of a SIMD architecture with a memory core. The generic C•RAM architecture proposed by Elliott is presented as a reference platform. It provides the basis for comparison with the implementation of C•RAM that is used in this work, the Accelerix AX256 graphics accelerator.

Section 2.3, Controllers for SIMD architectures, looks at previous SIMD controllers and presents a classification system for controllers. Two controllers for C•RAM devices are examined: the controller for the AX256 and a research controller for a prototype implementation of C•RAM.

2.1 SIMD Architectures

The term Single Instruction stream Multiple Data streams (SIMD) was introduced by Flynn in 1966 [7]. He presents a classification of computer architectures based on the manner of instruction and data distribution. The four types he identifies are:

- Single Instruction stream Single Data stream (SISD)
- Single Instruction stream Multiple Data streams (SIMD)

- Multiple Instruction streams Single Data stream (MISD)
- Multiple Instruction streams Multiple Data streams (MIMD)

The two classes for existing parallel architectures, SIMD and MIMD, represent a difference in instruction distribution. In a SIMD architecture all the processors execute the same instruction on multiple data items at once. In contrast, within a MIMD architecture each processor obtains its instructions separately. Systems with multiple microprocessors such as an SGI Origin are examples of the MIMD architectural type.

This classification system, while straight forward, does not provide detail about the type of parallelism utilized or the applications suitable for a particular parallel architecture. A classification system for parallel architectures used by Sima, Fountain and Kacsuk [8] divides parallel architectures into data–parallel and function–parallel types. Some architectures, however, possess characteristics that could labeled as either type. A MIMD architecture is one example of parallelism in both data–parallel and function–parallel terms. In these cases significance is given to the function–parallel characteristics, and the architecture is labeled as such. SIMD is classified as a data–parallel architecture due to its single instruction stream, but multiple data streams. This type of parallelism signifies that SIMD architectures are best suited for computation on arrays of data common in applications such as signal processing.

For applications that can make use of the architecture, SIMD provides a parallel processing solution at a better performance/cost ratio than a MIMD implementation. The absence of multiple instruction streams and their associated complexity means that a SIMD machine can have a large number of processors in its array at a lower cost than MIMD. In order to facilitate the implementation of a large number of processors, the computational units in a SIMD array can be reduced in complexity to achieve a higher density of processors within a unit area. The loss in performance per processor is made up by the increase in the number of processors that can be implemented in the same space. A SIMD architecture with simple bit-serial processors can implement fine grain parallelism by assigning a single data item to each processor. The SIMD computer developed by Thinking Machines, the Connection Machine 2, takes advantage of this to offer 65,536 single bit-wide processors. This is in contrast to early SIMD machines like the Illiac IV, which has 64 64-bit processors.

The relative simplicity of SIMD architectures when compared to MIMD allows SIMD to claim some advantages [8]:

- simplicity of concept, programming, and synchronization
- regularity of structure
- easy scalability of size and performance
- straightforward applicability in a number of fields which demand this type of parallelism.

MIMD architectures, while more complex, are capable of speedups on a larger set of algorithms than SIMD architectures are. SIMD architectures, however, can be implemented with a better performance/cost ratio and have seen success when coupled with a host system to create a computing platform with economical parallel processing. In such a system the host can off-load data-parallel functions to the SIMD array, while sequential code can be executed on the host [9].

The common composition of a SIMD machine, illustrated in Figure 2.1, consists of an array of processors, also called processing elements (PEs), localized memory as data storage, an array controller, a host computer, and I/O for instructions and communication. The array controller sits between the host and the SIMD array to handle instruction issuing and data communication. The specifics of this role vary according to the particular SIMD architecture. In order to maintain some control over the execution of instructions, the array controller might mask or disable one or more processors. In an architecture with some local autonomy a processor might disable itself from execution based on data in its local storage. This could extend to allowing the PEs to determine their individual memory addressing when accessing the local data storage (non-uniform addressing). Otherwise the array controller provides a common memory address to all PEs when the local data storage is accessed

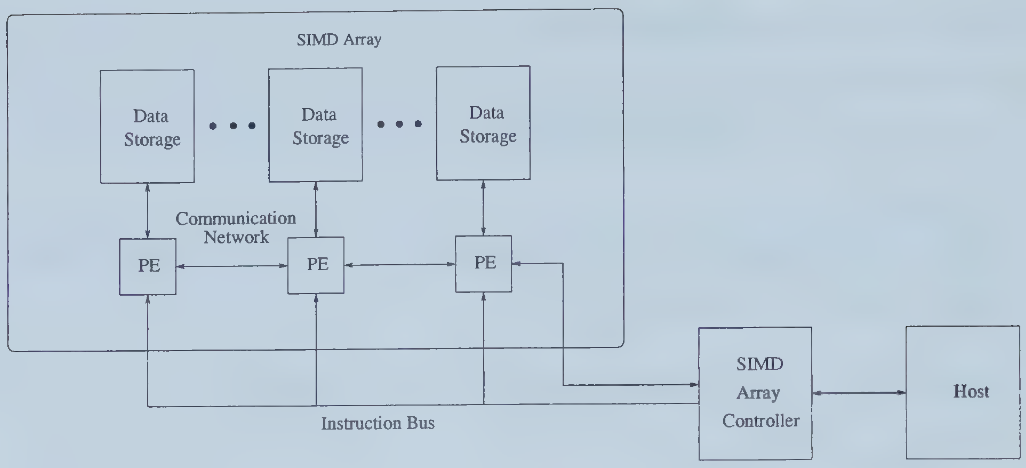

Figure 2.1: A SIMD Machine

(uniform addressing). The features of SIMD architectures vary based on intended use.

There are many possible variations in architecture, and some SIMD machines are designed for a particular application. This reduces the complexity and the associated cost of a SIMD machine, which was an important factor in early computers. The Ballistic Missile Defense Agency's Parallel Element Processing Ensemble (PEPE) is a SIMD machine that was used specifically with a CDC 7600 for radar processing. NASA's Massively Parallel Processor (MPP) was used for processing satellite images, a job it did well [10]. These early SIMD machines, though application specific, demonstrated the capabilities of SIMD.

Later SIMD computers, such as Thinking Machines' CM series and MasPar's MP series in the 1980s and early 1990s, targeted a wider range of applications in scientific computing. These machines, however, suffer from limited functionality, low processor utilization on some problems, and unsuitability for many important problems [9] [5]. Their high price meant they served a niche market of universities and government agencies which could afford them.

Recent years have seen SIMD architectures emerge into mass market application as platforms for multi–media, as a result of improvements in semiconductor technology. This has happened in two forms: small–scale SIMD architectures integrated into microprocessors, e.g. Intel's Pentium Matrix Manipulation eXtensions

(MMX) [11] and AMD's 3D-Now [12], and digital signal processors (DSPs) [13] [14]; and SIMD architectures being implemented around memory cores [3] [15] [16]. Small-scale architectures segment ALU operations into a small number of parallel operations. For example, in the MMX architecture a 32-bit word of data is broken into four 8-bit chunks and the same operation is performed on all four chunks.

SIMD architectures implemented around memory cores differ significantly from small-scale architectures in their level of parallelism. These SIMD architectures are massively parallel. The architectures are designed to take advantage of the high internal data bandwidth by placing SIMD processors inside memory devices. The combination of high data bandwidth and the fine grain parallelism possible with a SIMD architecture allows data-parallel operations to be performed in memory. These processors in memory can be added to memory designs at low expense because of their SIMD architecture [1], thereby providing an economical parallel processing solution.

2.2 C•RAM Architecture

2.2.1 Overview

Computational RAM (C•RAM) is part of a group of technologies known as logic-enhanced memories or processors in memory (PIM) systems. The common theme among these systems is the implementation of computational units alongside memory cores inside chips. This allows one or more operations to be performed without moving data off-chip. An early paper by Stone [17] in 1970 noted the possible benefits of integrating memory and computational units inside a single device, but specific architectures did not arrive until much later.

C•RAM, developed at the University of Toronto [5], integrates a SIMD array of processors with a SRAM or DRAM memory core. Its PIM architecture places processing elements at the sense amplifiers of memory columns, taking advantage of the estimated 2.9 Terabytes/s bandwidth that is available at the memory columns [1]. As a SIMD architecture, C•RAM implements fine grain parallelism by using

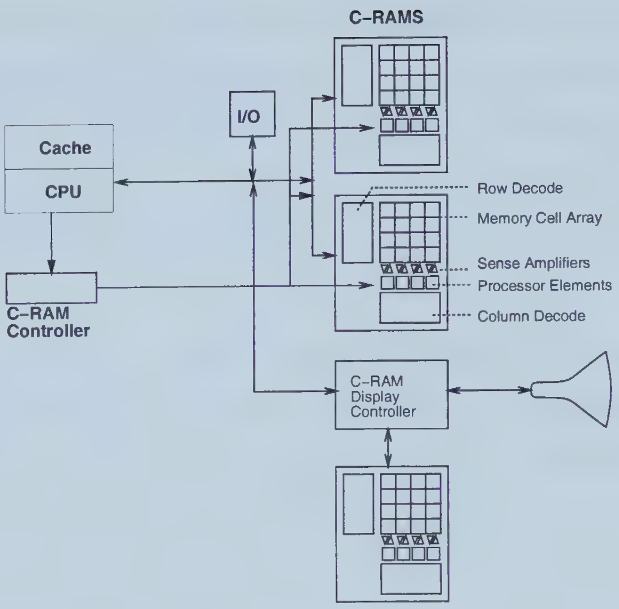

Figure 2.2: A System Architecture with C•RAM

bit-serial PEs. The C•RAM PEs are built around a set of registers and a 256 function ALU that uses an 8-bit opcode generated off-chip. The PE ALU differentiates C•RAM from application-specific logic enhanced memories that implement a small set of fixed functions [15]. This flexibility in the C•RAM PE allows it to appeal to a more general set of applications that are data-parallel. Some of these applications have been identified in prior research [1] [4], including signal processing, vector quantization, data mining, and fault simulation.

C•RAM's architecture performs well with data-parallel operations, but like other SIMD systems it requires a host to handle all other processing. C•RAM is designed to be integrated with a SRAM or DRAM memory core and maintains the conventional memory interface. The intention behind this design is to allow C•RAM to be used in place of ordinary memory in a computer system, as illustrated in Figure 2.2 [1]. A C•RAM device can be treated as normal memory when accompanied by support logic in the form of a C•RAM controller.

When compared to normal memory, a C•RAM implementation adds an increase of 3% to 20% in area and a power overhead of 10% to 25% to a memory core [5]. DRAM memory is widely used in computer systems. For such systems C•RAM devices can offer a massively parallel processing capability at a low cost/performance

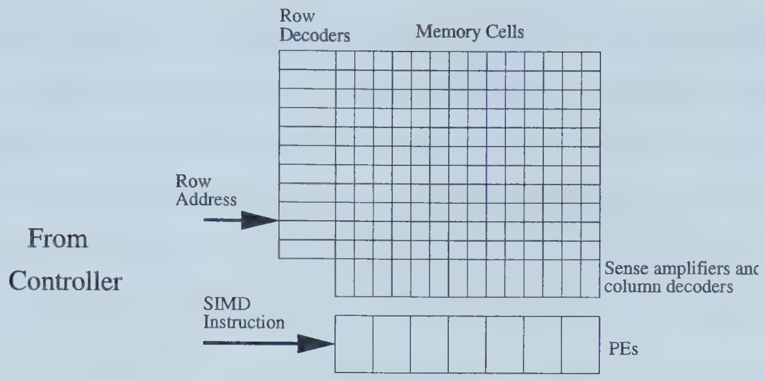

Figure 2.3: Simplified C•RAM Architecture

ratio.

C•RAM has been implemented by universities in research prototypes with 64 [18] to 512 PEs [19]. Accelerix Inc.[1] produced a commercial variant, the AX256 graphics accelerator, which implemented a C•RAM architecture with 4096 PEs and 13.4 Mbits of DRAM [3]. The AX256 is the C•RAM device used in this work, as a result its architecture is discussed later.

2.2.2 Generic C•RAM Architecture

The C•RAM architecture described by Elliott [5] builds on the structure used by DRAM and SRAM memory cores. The design of C•RAM attempts to minimize the architectural change required in the memory device to implement C•RAM, and in the process minimize the cost of implementation. C•RAM can be implemented in a DRAM IC process without the need for additional steps to accommodate the logic circuits of the computational units.

The following sub-sections divide the discussion about the C•RAM architecture into three parts: memory organization, processing elements, and host interface.

Memory Organization

The memory core in C•RAM has memory cells organized in an array of rows and columns. This organization is standard in DRAM and SRAM memories. When a row of cells in the core is enabled, each cell connects to the respective bit-lines for

[1]Accelerix Inc. was acquired by Mosaid Technologies in May 1999

its column. The bit-lines connect to sense amplifiers at the base of each column that amplify the logic value on the bit-lines. To carry data off–chip, a data bus connects the sense amplifiers to the external data interface. The sense amplifiers are connected to the data bus in blocks. A column decoder determines which block of sense amplifiers is connected to the off-chip data bus for each access.

Even though the entire row of memory cells is enabled, only sections of the row equivalent to the data word are read off-chip. For a memory chip that has a data word of just 32 bits and a row of 4096 memory cells, only 0.8% of the possible bandwidth is used. This unused bandwidth is what C•RAM capitalizes on by placing its PEs at the sense amplifiers of the memory columns. For optimal usage of space, the placement of PEs is pitch matched to small multiples of the columns of memory cells. The memory technology used, either SRAM or DRAM, affects how many columns are matched to a PE. With SRAM, which uses larger memory cells than DRAM, the ratio for matching might be 1:1 or 1:2 for PE to memory column. For DRAM with its smaller cells, this can be higher, e.g. 1:4. Normally only one column at a time is connected to the PE during memory operations, using a mux/demux circuit as needed.

The placement of the processing elements at the sense amplifiers does not interfere with the host's access into the memory core of the C•RAM device. The host can still access the memory core like an ordinary implementation of that memory technology. This means that a DRAM–based C•RAM device can be accessed like an ordinary DRAM device. This allows the sharing of memory between the host and the C•RAM PEs, and prevents a bottleneck from occurring as a result of data being moved in and out of C•RAM memory. Instead, data can remain in the C•RAM memory and remain accessible to both the C•RAM PEs and the host processor.

C•RAM Processing Element

The C•RAM processing element, Figure 2.4 [5], is a bit-serial unit built around a multiplexer–based ALU, a set of registers, and a memory interface. In order to keep the cost of implementation low, a register file is not used in the PE. Access

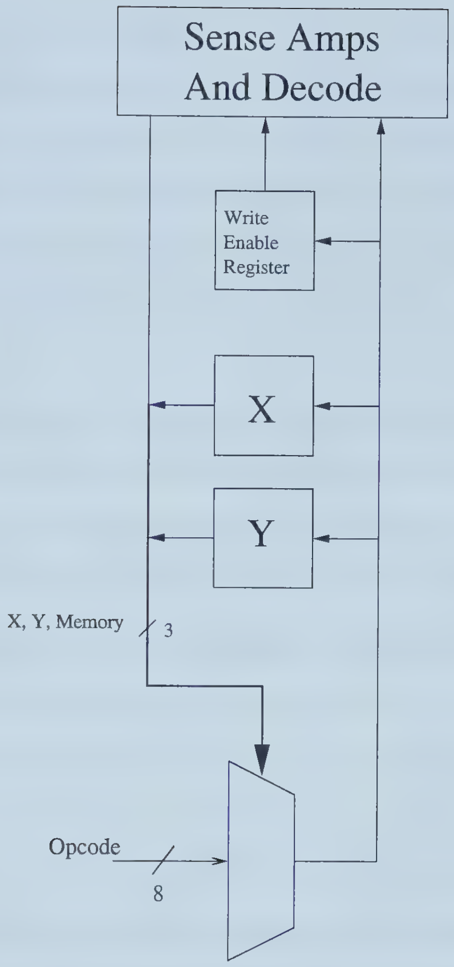

Figure 2.4: C•RAM Processing Element

into memory is performed using uniform addressing, in that all the PEs access the same row of memory during memory operations.

The ALU is an 8-to-1 multiplexer that can be used to perform one of 2^8 operations on three operands. The X and Y registers are used as two of the operands, and a memory location is used as the third operand. The 8-bit opcode is broadcast from off-chip to all PEs and is used to perform one of 256 boolean functions on the three operands. The result of the operation can be saved to either the X or Y register or written back to memory. For conditional operations, a *write enable* register is used to control whether the result of the operation is written back to memory. The value of this register is determined by the result of a previous ALU operation, allowing conditional operations to be affected by data local to the PE.

The 256-function ALU is chosen over other feasible ALU implementations because it has the best performance/cost ratio. This evaluation is based on the ability to perform vector addition, which forms the basis of other operations including multiplication. The 256-function ALU is used twice to perform this operation: first to calculate the sum, and then to calculate the carry. This can be repeated to add multiple variables in parallel if each individual variable is stored in the column of a single PE.

The organization of variables in the columns of the PEs is indicative of a difference that can occur in the orientation of data for the PEs and the host processor. If the data stored in the C•RAM device is orthogonal to the format expected by the host system, then the data must be rotated when it is retrieved by the host. This can be performed either in the host's software or in corner-turning hardware such as a specialized cache.

Communication

For communication between PEs, C•RAM uses a network composed of two parts: a broadcast bus that can also be used for combine operations, and a scalable-grid communications network with 1-D communication between rows of PEs and 2-D communication among C•RAM chips. Its design is intended to be compatible with DRAM and to use less IC area than the non-communication logic of the PEs. The network implements three types of communication: point-to-point communication between a single sender and receiver, broadcast communication from a single sender to multiple receivers, and combine operations that combine the outputs of multiple PEs.

The broadcast bus is implemented as a Wired–AND bus arranged in a tree of bus segments with repeaters to increase drive and reduce delays. The repeaters can be selectively disabled to provide localized communication between PEs or enabled to provide a global bus. By controlling the number of PEs that write to it, the broadcast bus can be used either to broadcast data to multiple PEs or to perform a combine operation on the outputs of a set of PEs.

The scalable-grid network, Figure 2.5 [5], operates in two modes: 1-D commu-

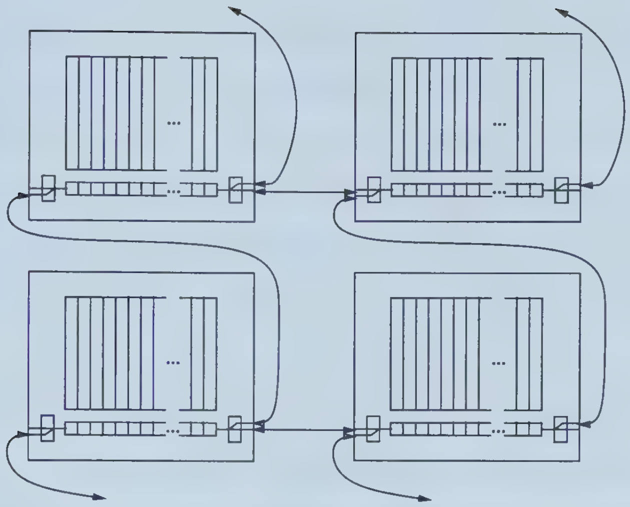

Figure 2.5: C•RAM Scalable-Grid Network

nication along a row of PEs, and 2-D communication among C•RAM chips. The 1-D communication is implemented as a left-right shifting along a row of PEs. In this manner point-to-point communication can be performed by using the required number of hops to transport data from the sender to the receiver. For inter-chip communication, a 2-D network is used to communicate between multiple C•RAM devices allowing both vertical and horizontal communication. To control the direction of data, switches are used to route communication along the horizontal or vertical portions of the grid.

The capabilities of each part of the communication network as well as the required bandwidth determines how they are used to implement the three communication functions. Figure 2.6 [5] illustrates a PE with the circuit features used to implement the C•RAM communication network.

External Communication

For I/O transfers between the C•RAM device and external devices, including a host, communication can occur over a memory bus. However if only a single memory bus is used, bandwidth does not scale with the number of C•RAM chips and PEs. To address this, Elliott [5] proposes a two step staggered approach to implementing external I/O communication. The first step is to connect the external I/O to the interprocessor communication network of the C•RAM devices. For implementations

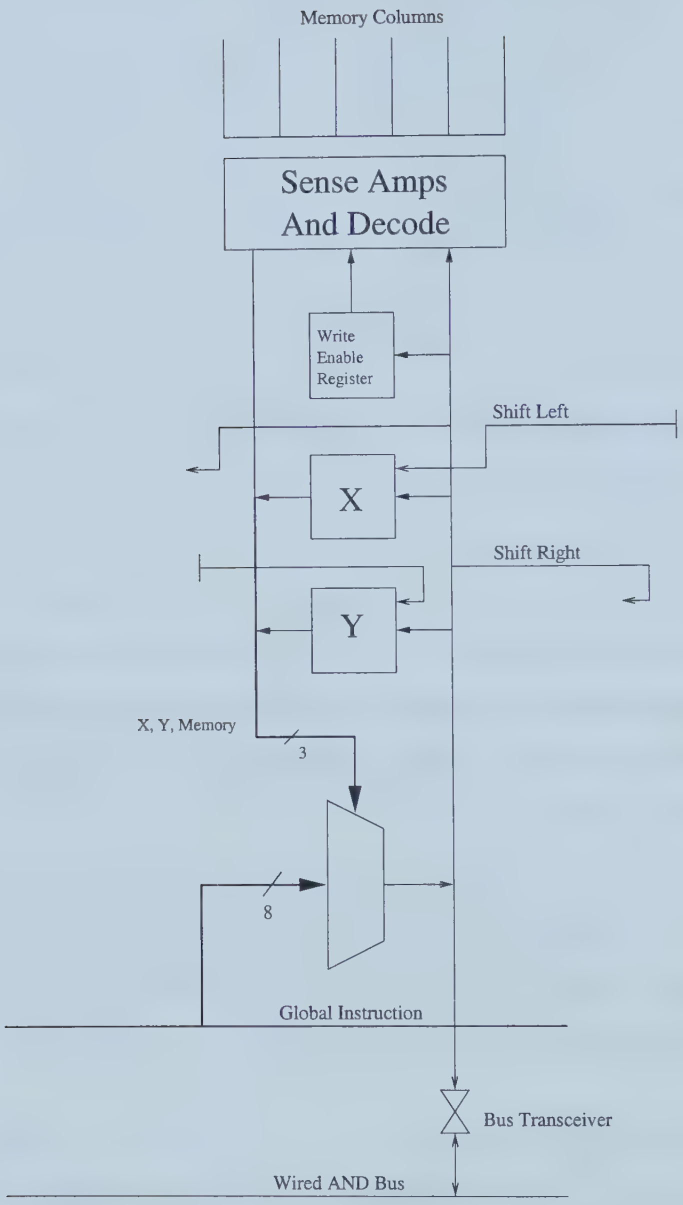

Figure 2.6: C•RAM Processing Element with Communication Interfaces

that require even higher bandwidth, a video-RAM (VRAM) port could be added as an additional step, but would carry a penalty in silicon area.

The existence of the scalable left-right interprocessor communication network between devices provides tap points at the edges of C•RAM chips that could be used for implementing I/O ports. The location of I/O ports can be adjusted for the level of bandwidth desired. By placing them at the perimeter of the 2-D grid of C•RAM devices, higher bandwidth than with a single memory bus can be realized. Additional bandwidth can be obtained by placing I/O connections at each chip, which would enable concurrent I/O to be performed with each C•RAM chip.

In applications where the interprocessor communication network does not provide the necessary bandwidth, a video-RAM (VRAM) port can be added to the C•RAM device. VRAM is designed to provide a continuous stream of data without the need to maintain a set of elastic buffers. By using banks of sequential access memory, a VRAM implementation can engage in data transfer using one bank while the contents of another is being transferred either from or to memory. A VRAM port will appeal only to applications that require the additional bandwidth, because it carries a penalty in silicon area.

2.2.3 Accelerix AX256

Accelerix Inc., acquired by Mosaid Technologies, implemented the first commercial variant of C•RAM, the AX256 graphics accelerator [3] [20] [21]. The device is DRAM-based with 4096 PEs and 13.4 Mbits of memory storage. As a C•RAM device, it does not implement all the features of the generic C•RAM architecture described in the previous section. The particulars of its design and how it varies from the generic C•RAM architecture are discussed in this section.

Memory Organization

The memory organization of the AX256 is similar to that of the generic C•RAM architecture. The memory core of the AX256 is DRAM and is laid out in a grid pattern. It is accessed using a combination of a Row Address Strobe (RAS) signal and a Column Address Strobe (CAS) signal. The RAS signal is used to enable the decoding of the row address to select a row and the CAS signal is used to enable the decoding of the column address to select a data word within the row. In this manner it operates similarly to ordinary DRAM, as specified by the C•RAM architecture, but with a synchronous interface that is described later. The smaller size of the DRAM cells compared to the area of a PE requires the pitch matching of multiple memory columns to each PE to remain space efficient. In the case of the AX256, four memory columns are pitched matched to each PE.

The memory hierarchy at the highest level exists as eight banks. Each bank has a set of 512 PEs, which amounts to a total of 4096 PEs per device. Multiple banks can be enabled at the same time to allow SIMD operations to span more than one bank. Each bank consists of 816 rows of memory cells. A row is separated into four segments, of which only one can be active at a time. This reflects the pitch matching of four memory columns to each PE. Each segment is composed of sixteen words, 32 bits in width. Figure 2.7 [21] illustrates this memory hierarchy.

Processing Element

The AX256 PE extends the C•RAM PE to include five registers, but still uses an 8-to-1 Mux as the ALU. Three of the registers BRUSH, SRC, and DEST, are used

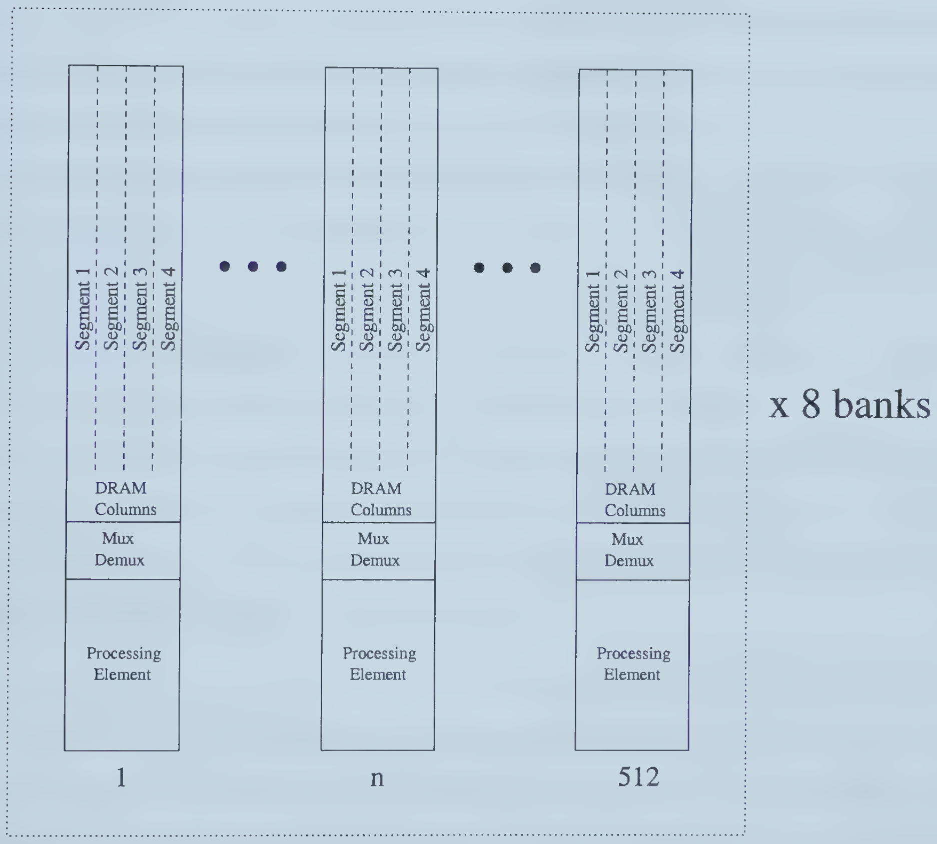

Figure 2.7: AX256 PE and Memory Organization

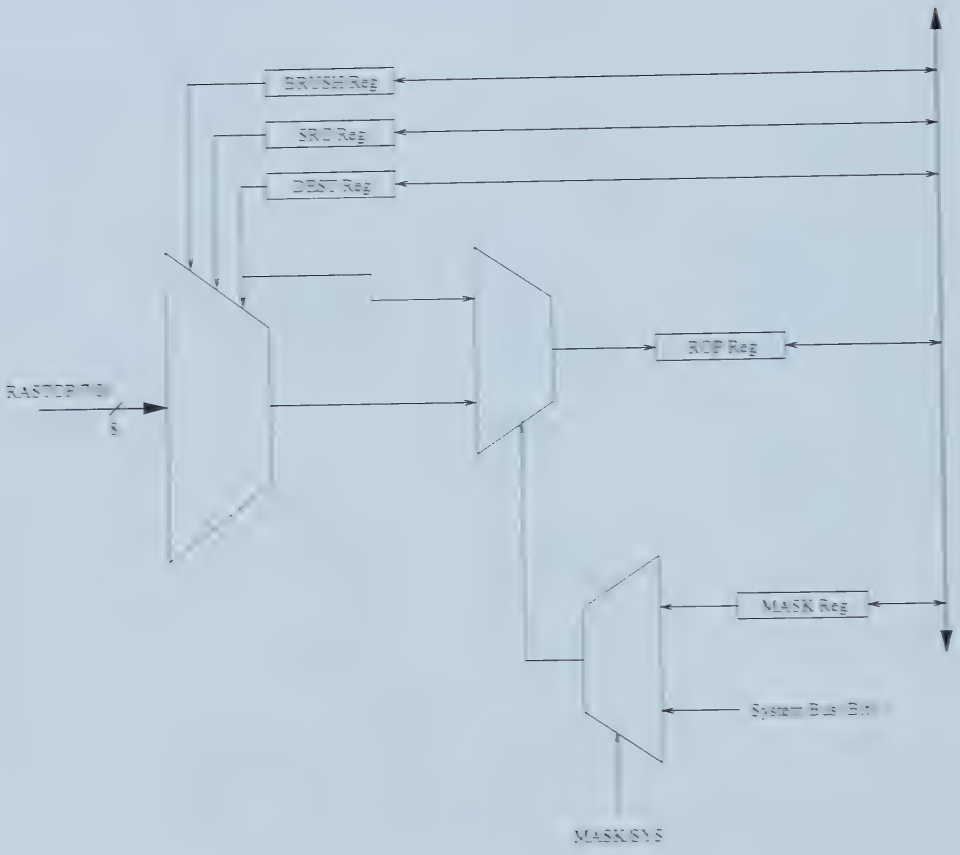

Figure 2.8: AX256 Processing Element

as inputs to the 8-to-1 Mux. The generic C•RAM PE also has three operands, but only two are registers. The third is a memory location. Instead of a memory location, the DEST register in the AX256 PE serves a similar purpose and its value is passed through if the PE is masked during opcode execution. A mask bit determines whether the result of the operation or the value of the DEST register is saved. The mask bit can come from one of two sources: the MASK register, or the appropriate bit of the system data bus which interfaces with the host. The value of the MASK/SYS signal determines which source is used for masking the operation, and the final result is stored in the ROP register. Figure 2.8 [21] illustrates the organization of the AX256 ALU.

Like the memory core, the PEs are allocated in banks. Each bank of memory has a corresponding set of 512 PEs. When a bank is enabled the PEs associated with that bank also become active. If all the banks are enabled, then all PEs in the AX256, 4096 in total, are active. Each PE has access to a memory column in each of the four segments in a bank, and the bits in these columns serve as the local storage for each PE. By using bank enabling and disabling, and masking through the mask bits, conditional operations can be configured for coarse-grain or fine-grain exclusion of PEs. Bank control affects data movement as well as opcode execution, while mask bits only affect opcode execution.

Data values can be moved between PE registers and the DRAM memory core in word sizes of 32-bit, or the full 512 bits per bank (1 bit per PE). In this manner, a row of data can be loaded from the selected segment in the DRAM memory core into the PE registers in one parallel load. Unlike transfers between the DRAM memory core and the PE registers, movement of data between the PE registers is performed only for a complete row, i.e. 512 bits per bank.

Communication

The communication between PEs in the AX256 is conducted using a 32-bit funnel shifter, Figure 2.9. The shift left–right network and the broadcast network described in the generic C•RAM architecture are not implemented in the AX256. Combine operations of PE outputs as well as broadcast operations are not possible with the

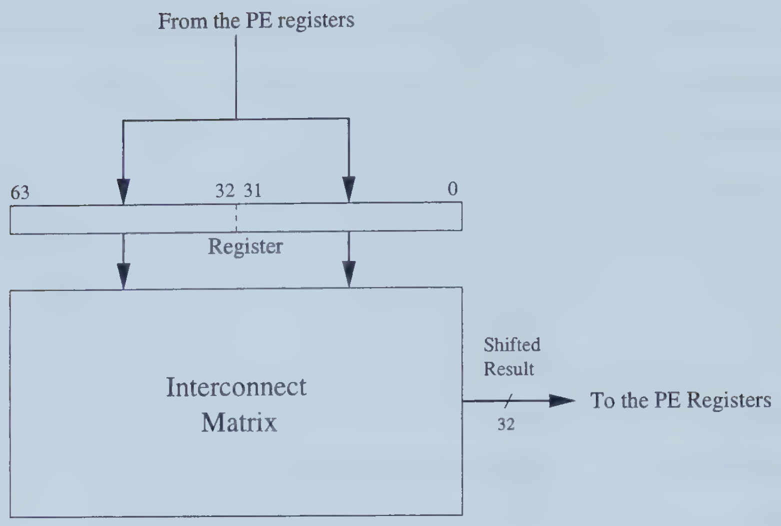

Figure 2.9: AX256 Funnel Shifter

AX256, because of the absence of a broadcast network. The funnel shifter, however, enables point-to-point movement of data between PEs.

The funnel shifter consists of a 64-bit register and a crossbar matrix for shifting. The register is divided into two parts: a high section (63:32) and a low section (31:0), which can be loaded simultaneously with the same value or loaded separately. When data is being shifted, the crossbar matrix performs a shift of up to 31 bits on the 64 bit register. The higher or lower portion of the result can then be written back to a 32-bit word in one of the PE registers. Each bank has one funnel shifter associated with it. The funnel shifter can shift data around the row of PEs in a bank, because it can store two 32-bit words and shift the combination of the two words.

External Interface

The AX256 has an on-chip controller that connects to the host system through a PCI bus interface. For the purposes of this work, however, the AX256 is run in a special test mode that disables the on-chip controller and allows an external controller to access the C•RAM architecture. This test interface is the one described in this section.

The memory interface of the AX256 is similar to that of an ordinary DRAM device, Figure 2.10. It is composed of two bus interfaces: a 10-bit address bus,

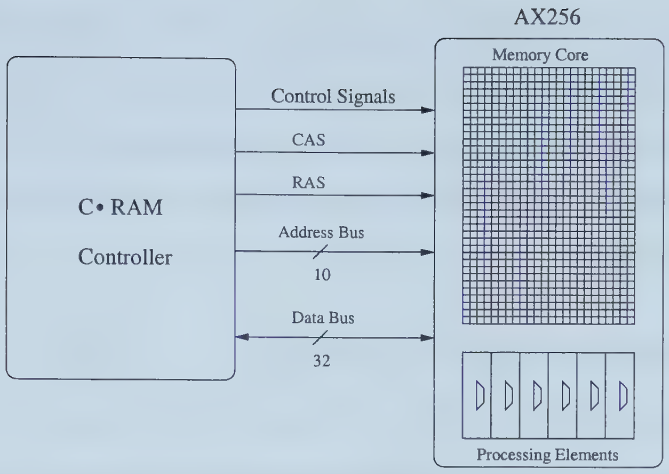

Figure 2.10: Controller – AX256 I/O Communication

and a 32-bit bidirectional data bus. The address bus is multiplexed between row addressing and column addressing with addresses loaded in a "row then column" sequence. The latching of the row and column addresses is controlled by the RAS and CAS signals from the controller. These signals, RAS and CAS, are also used to enable row address decoding and column address decoding. Using the address and data bus, the controller can access both the memory core and the C•RAM SIMD architecture. The interface is synchronized by a clock that is generated by the external controller.

Additional Features

The AX256 has additional features that support its operation as a graphics accelerator such as a Serial Output Register (SOR). As these features are specific to the role of a graphics chip and not influential in its performance as a C•RAM device, they are not discussed here.

2.3 Controllers for SIMD Architectures

SIMD architectures provide a high performance computing platform for data-parallel applications, but require a host system to execute sequential programs. It is the combination of a SIMD architecture with a host system that provides the optimal platform for applications. Communication between the host system and the SIMD array is managed by a SIMD controller. The controller can be responsible for managing the SIMD data storage and other periphery tasks, but is distinguished by how it streams instructions from the host to the SIMD array. The method used for streaming instructions varies among different SIMD machines and can also vary in its level of complexity.

2.3.1 Examples of SIMD Controllers

Microcoded Sequencer

The microcoded sequencer uses a set of microcode instructions to determine how the controller interacts with the SIMD array, including the issuing of SIMD instructions. This was the most common form of sequencer in past SIMD machines, and was used in the controller for the Illiac IV, the AMT DAP 500 [22], MIT Pixel-Parallel Image Processing System [23], and the Thinking Machines' CM-1 and CM-2 machines [24].

One method of using the microcode instruction set of the sequencer is to implement a library of functions that perform small-grain operations, e.g. add 32 bits, or more complex functions such as a fast fourier transform (FFT). This library is stored on the controller, and the functions are called by host during program execution. An example of this is the PARIS language for the Thinking Machines' CM SIMD computers.

The implementation of a function library is one method, but machines such as the Illiac IV used a different approach. The program code for the Illiac IV was compiled on a Burroughs B-6700 and then transferred to the Illiac IV. Rather than use the small grain approach of a function library, this method is at the opposite end of the granularity scale. Application development focuses on programs for the SIMD array rather than the use of a function library implemented on the array.

The AMT DAP 500, a machine capable of using both methods, had a controller that was capable of scalar operations. Its controller could be used with programs that included scalar operations, or with a function library that could be called by the host. The difference is the level of granularity in the method of operation.

Even though it is the most common implementation for SIMD controllers, it appears that controllers based on a microcoded sequencer suffer from low PE utilization. Evaluations of a Thinking Machines CM-2 executing a boolean parallel application revealed a PE utilization of less than 1 %. It is believed that overhead in the communication between the host and the SIMD array resulted in such a low utilization [5].

Co-processor Sequencer

The co-processor sequencer is the approach used by MasPar in the MP SIMD computers [25]. The controller is a dedicated RISC processor that is capable of executing sequential code in a manner that allows a complete application to run on the SIMD computer. The data-parallel portion of the executing program is performed by the SIMD array and the controller executes the sequential portion of the application. The user interface for programs executing on the SIMD computer is still implemented on the host system.

Host–Based Sequencer

The host–based sequencer is the approach taken by the Terasys PIM machine [26]. Moving away from the microcoded sequencer, the Terasys machine depends on the host system to issue SIMD instructions and uses an interface board to handle low-level communication between the host and the SIMD array. Using bits in its address field, the host indicates to the interface board whether it is accessing memory or issuing an instruction. To issue an instruction, the host uses a 12-bit index into a lookup table of 4096 25-bit SIMD opcodes on the interface board. This table is loaded by the host during program initialization and allows it to implement a select set of opcodes out of 2^{25} possible functions.

2.3.2 C•RAM Controllers

As a SIMD architecture, C•RAM requires a controller to interface with the host system or processor. As a PIM technology, C•RAM is designed to work in a mode compatible with ordinary memory, and a C•RAM controller has to include logic to support the memory core in addition to the functions of a SIMD controller. This is more prevalent for DRAM rather than SRAM implementations, because of the refresh requirements of DRAM. As DRAM has a lower cost per cell than SRAM and is the more common form of computer memory, it is expected that most C•RAM controller implementations will have to include refresh support.

The functionality of the C•RAM controller is divided into three parts:

- Sequencer – For streaming SIMD instructions to the array of processors.
- R/W Access – To allow the host to access the C•RAM memory core.
- Refresh – To handle the refresh of DRAM memory

For SRAM implementations of C•RAM there is no need for a refresh circuit, but the other two components must be present.

As the multiple functions of the controller require access to the C•RAM device, timing and priority circuitry is required to manage access. The arrangement used to determine queuing status is dependent on the implementation and application intended. On a DRAM device refresh would normally be given priority access, as this is required to prevent data loss. The ordering of host memory access and sequencer access, however, will be dependent on whether fast memory access is a priority, or whether the priority is the rapid execution of SIMD instructions.

In addition to managing priority, the C•RAM controller has to manage timing to prevent race conditions between instructions and data. This is highlighted in implementations where the sequencer can execute a sequence of microinstructions based on a function call within a library. In these situations it is possible that the host might attempt a memory access that affects a data value used in the executing sequence of SIMD instructions. To prevent such hazards from occurring, the C•RAM controller has to manage the time line for both memory access and instruction issue.

In the following discussion, two C•RAM controllers are mentioned: the on-chip controller for the AX256 graphics accelerator, and a controller designed by Peter Nyasulu for a C•RAM research prototype.

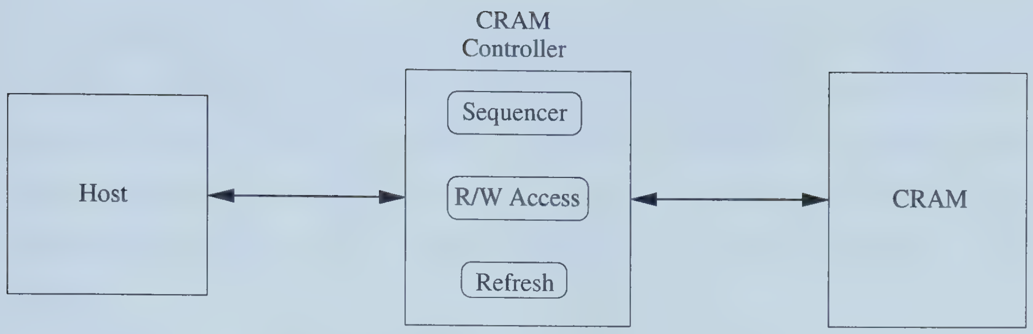

Figure 2.11: A Generic C•RAM Controller

Accelerix AX256 Controller

The on-chip controller for the AX256 graphics accelerator uses a host–based sequencer. For functions related to BITBLTs and PATBLTs, which involve the movement of data inside the memory core, the controller generates the necessary SIMD instructions. For other instances, the host can implement custom functions by issuing SIMD opcodes through the controller to the array of processors [27]. Designed to operate across a PCI bus, the host communicates with the controller through a series of registers that can be used for accessing memory, calling microcode functions, or issuing SIMD opcodes. The controller processes these accesses in the order that they are received, eliminating concerns about race conditions between instructions and data.

The AX256 controller is a proprietary design and details about its performance have not been published.

A Research C•RAM Controller

This controller designed by Nyasulu [28] for the SRAM–based C64p1k C•RAM device [19] is based on a microcoded sequencer scheme, and is illustrated in Figure 2.12 [28]. Intended to work with the C•RAM compiler developed by Elliott [29], the C•RAM assembly required by the compiler can be implemented on this controller using microinstructions.

The microinstructions are placed in a re-writable code store and represent functions related to various operations, including those required by the C•RAM compiler. The functions are called by mapping host instructions to specific locations in

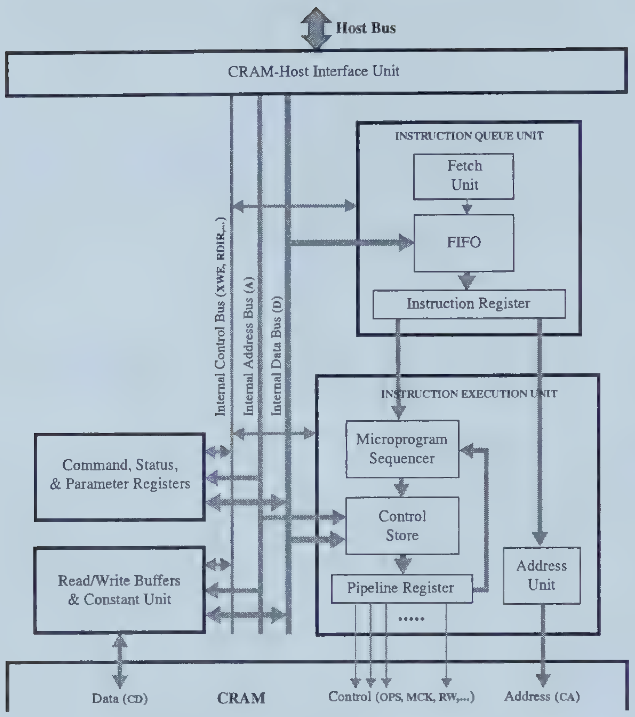

Figure 2.12: Peter Nyasulu's C•RAM Controller

the code store where the appropriate sequences of microinstructions resides.

Instructions from the host are placed in a FIFO for processing by the sequencer and are treated separate from data. Instead, data is transferred using read/write buffers that allow the host to preload data if the sequencer is in the process of executing a function. To prevent race conditions between data and instructions, an instruction to transfer data is inserted into the instruction queue by the host. When the sequencer retrieves the "data move" instruction, it transfers the associated data either as a read or a write.

While this controller was implemented in hardware [28], no performance results have been published and the state of its functionality is not known.

2.4 Summary

SIMD provides an economical parallel processing platform that can be used with data–parallel algorithms. When combined with a host, it provides a computing platform for applications that combine sequential and parallel functions. C•RAM is a processors-in-memory technology that integrates a SIMD architecture with DRAM or SRAM memory in a chip. Its design maintains the conventional memory interface used by DRAM and SRAM. When integrated into a system, C•RAM can be used as ordinary memory and a massively parallel processing architecture, when required.

The C•RAM controller enables the integration of C•RAM devices into a system. Like controllers for previous SIMD architectures, the C•RAM controller manages the communication between the host and the SIMD array. The existence of the memory core within the C•RAM architecture requires that the C•RAM controller combine the role of a SIMD controller with that of a memory controller.

Chapter 3

C•RAM Controller

The integration of C•RAM into a system architecture is dependent on a controller that enables communication between the host and the C•RAM device. Through the controller the host can access both the SIMD parallel architecture, as well as the memory functions of the C•RAM device. In the past, many SIMD controllers have been based on a microcoded sequencer that expands macro instructions into sections of microcode. As previously mentioned, the Terasys SIMD machine takes a different approach for its controller. It allows the host to directly issue SIMD instructions, using the controller to facilitate low-level communication between the host and the SIMD array.

The C•RAM controller we propose is similar to the Terasys SIMD controller. The ability of C•RAM to be used as a replacement for ordinary DRAM or SRAM memory leads to the expectation that C•RAM will be used as part of a system's memory. In the hardware platform used to implement our controller and most expected implementations, the host will be a microprocessor that interacts with the various devices in the system. Significant improvements in the capabilities of microprocessors have provided resources that can be used to implement a C•RAM controller. In the background chapter (Chapter 2) the role of the C•RAM controller is defined in three parts:

- Sequencer – For streaming SIMD instructions to the array of processors.
- R/W access – To allow the host to access the C•RAM memory core
- Refresh – To handle the refresh of the DRAM memory core

In our proposed controller the task of "sequencer" is assigned to the host processor, and the controller manages the low-level communication necessary for the host to issue SIMD instructions to the C•RAM device. The two other tasks remain the responsibility of the controller. This type of controller has characteristics that we believe are important:

- **Minimal Hardware** – The amount of hardware required to implement the C•RAM controller will have a direct impact on how feasible it will be to integrate with other circuits. Implementation on a single device with a C•RAM architecture or with a microprocessor in a system-on-chip (SOC) are possible end results for a controller design. In these situations, a controller with complex hardware can be quite challenging and not necessarily feasible to implement.

- **Reasonable Access** – For the C•RAM device to integrate well into the system architecture it has to be reasonably accessible by the host, as well as other devices that might not necessarily be aware of the C•RAM architecture. System components, such as Direct Memory Access (DMA) controllers, could have a strong bearing on how well a C•RAM architecture performs within the system. In light of this, the C•RAM controller has to be able to accommodate access to the C•RAM memory by these devices and the host processor.

- **Low Overhead** – As communication between the host and the C•RAM device is conducted through the controller, a significant overhead in the controller becomes a hinderance to the performance of parallel processing. To achieve a reasonable level of C•RAM PE utilization, which acts as a measure of controller efficiency, the overhead introduced by the controller must remain low.

In section 3.1 we describe this controller and identify its components based on the experience gained from the implementation of our hardware platform. The Accelerix AX256 is used as a reference C•RAM architecture. This device is a commercial variant of C•RAM that contains 4096 processing elements and 13.4 Mbits of DRAM memory. As DRAM is commonly used as memory in many systems, we believe a controller for a DRAM-based C•RAM device such as the AX256 has wider application than one developed for SRAM-based devices.

To improve system performance, techniques such as caches and burst modes have been developed for DRAM access. With our focus on the interaction between the host and the SIMD architecture, we present enhancements to the C•RAM controller that can be used to improve system performance. In section 3.2 we propose the use of instruction buffers within the controller to allow the host to perform burst SIMD instruction issues. In section 3.3 the addition of a microcoded sequencer to the controller is proposed. A paper design, included in the appendices, describes a microcoded sequencer that can be implemented in a C•RAM controller.

3.1 The Host as The Sequencer

In many previous SIMD machines controllers were based on microcoded sequencers, which enabled the host to issue function calls to the controller. Operations such as a "add 32 bits" would be seen as a single instruction by the host. In the controller these operations co-relate to sections of microcode that would step the sequencer through the interaction with SIMD array, including instruction issues necessary to perform the requested function. In our system architecture, however, the microprocessor host assumes the task of sequencing instructions. This simplifies the C•RAM controller, and takes advantage of the flexibility and performance of the host's microprocessor architecture.

As the SIMD sequencer the host is now responsible for the issue of individual SIMD instructions to the array. This exposes the host to a finer level of granularity in executing functions than a microcoded sequencer would. Instead of being able to issue a single "add 32 bits" instruction, the host must now issue the individual SIMD instructions necessary to perform this function, for example bit-level adds. With the host assuming the task of sequencing instructions, the controller does not need to implement the complex logic required to interpret microcode. In the absence of this additional logic the controller reduces the latency it inserts into the instruction, and data streams between the host and the SIMD array. The evaluation of the hardware implementation of this controller architecture has shown PE utilization of over 50%.

The use of the host as the SIMD sequencer provides access to the many features of a microprocessor architecture, such as branch mechanisms and arithmetic units. These features can be used to offer flexibility in the sequencing of SIMD instructions. If the microprocessor in many systems remains an under-utilized resource, the implementation of the sequencer on the host results in minimal negative impact on system performance. In return, applications have access to a flexible SIMD sequencer with low-latency communication to the SIMD array.

With the host responsible for sequencing SIMD instructions, the SIMD architecture of the C•RAM device must be visible to the host. This is in addition to the access that the host must have to the DRAM memory core in order for the C•RAM device to function as ordinary memory. The controller provides this access by mapping the interface to the C•RAM device into the memory space of the microprocessor host. Access to the C•RAM device for both the SIMD architecture and the DRAM memory core becomes reads and writes to memory addresses for the microprocessor. Existing development tools for the microprocessor can be used to write programs that interleave SIMD instructions with normal sequential code, because access to the C•RAM SIMD architecture is through memory addresses. In addition, other devices in the system that have access to the microprocessor address bus, e.g. DMA controllers, are able to see the DRAM memory core through the address space assigned to it by the controller.

Based on our experience from implementing a hardware platform that integrates C•RAM into an embedded system, we have identified the components of the controller that is based on the methodology of "the host as the sequencer." Illustrated in Figure 3.1, this controller is composed of four blocks: the address decoder, the address path logic, the data path logic, and the control logic. The address decoder is used to validate any accesses into the address space used by the controller. Signals from the address decoder alert the control logic to the type of access being initiated. The address path logic and data path logic are used by the control logic to perform any interface translation that has to occur between host and C•RAM

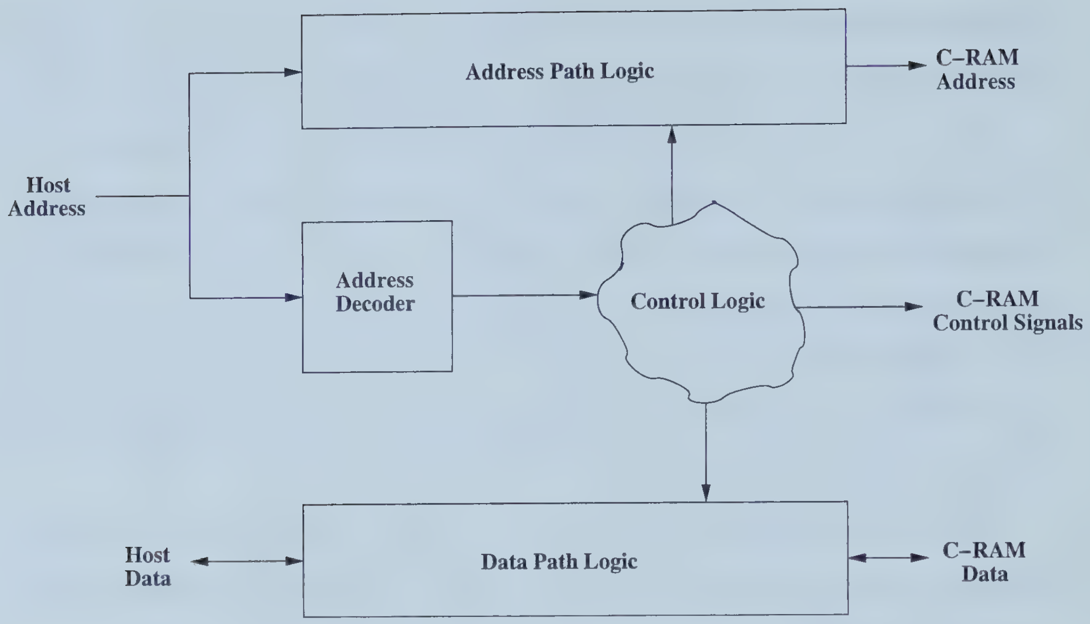

Figure 3.1: C•RAM Controller with a Host Sequencer

device. Handshaking with the host is the responsibility of the control logic. There is no handshaking with the C•RAM device, instead synchronous control signals generated by the controller are used.

Access to the C•RAM device is divided into regions of addresses by the controller. To access a particular function of the C•RAM device, the host reads or writes to the address allocated to that function. Reads and writes into the DRAM memory core are performed through a continuous region in the address space of the host. During this type of access the C•RAM device is perceived as ordinary DRAM memory. The continuous addressing also facilitates the use of DMA controllers and other similar devices.

The SIMD architecture of the C•RAM device is accessed via a register address model. Each SIMD operation is represented by a specific address and is activated when the host accesses that address. For example, to issue a SIMD instruction, the host writes the 8-bit opcode to the address assigned to that function. Other operations are performed in a similar manner.

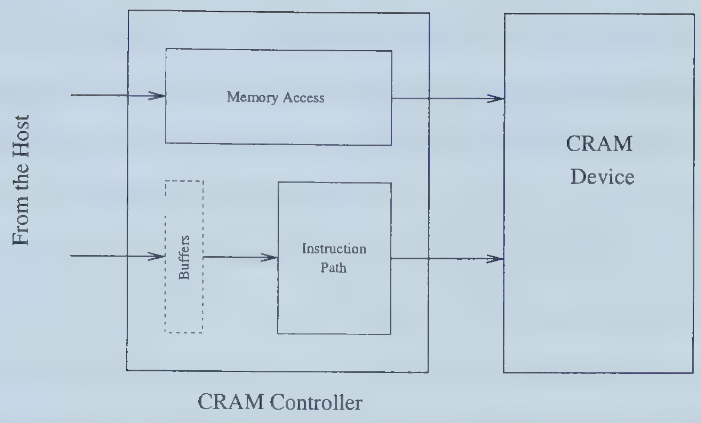

Figure 3.2: Inserting buffers into the instruction path of the C•RAM controller

Summary

In this section we presented a C•RAM controller that is based on a methodology of assigning the role of sequencer to the host. This reduces the complexity of the controller, and the associated latency that the controller inserts into the communication path between the SIMD array and the host microprocessor. Implementing the sequencer on the host takes advantage of the flexibility and performance of the microprocessor architecture. The controller provides access to the C•RAM device through the address space of the microprocessor, and applications executing on the host can access the SIMD architecture through reads and writes into this address space.

3.2 Buffering Access to The SIMD Architecture

In this section we present one of the enhancements proposed for the C•RAM controller: the insertion of buffers in the instruction stream. This enhancement is intended to reduce the load placed on the host during the issue of SIMD instructions. The use of buffers in the instruction stream permits the host to perform a burst transfer of SIMD instructions, as illustrated in Figure 3.2. The host is isolated from the latency of the C•RAM device and the host can attend to other system tasks, while the buffered instructions are being issued by the controller.

The use of buffers requires that some mechanism exists to coordinate data access with instruction issue, otherwise data corruption can occur if the memory core is accessed externally while buffered instructions are being executed. A simple solution is to "lock" the DRAM memory core by denying external access to the memory core while buffered instructions are being issued. In a system where the C•RAM device serves as main system memory this option is not viable, as such locks can exist for extended periods of time. Another and possibly more viable solution is to buffer reads and writes into the memory core alongside the instructions being issued, and maintain the order of arrival. Unfortunately both these solutions increase the latency of any memory access, though by differing amounts.

In addition to issues related to coordinating access to the C•RAM device, the impact of context switching by the host has to be taken into consideration. The implemented buffers have to be large enough to hold enough SIMD instructions to minimize any context switching performed by the host. Otherwise, the context switching by the host will eliminate any benefit gained from using the buffers.

As not all systems will benefit from using buffers in the controller, the inclusion of buffers in a C•RAM controller will be determined by the intended use of a system and whether a benefit is recognized.

3.3 Adding A Sequencer to The Controller

In the previous section the concept of using buffers was introduced as a method of reducing the amount of time required of the host to issue a block of SIMD instructions. This method depends on the host issuing a stream of instructions to the C•RAM controller, which buffers the instructions and then passes them onto the SIMD array. In this section we present a second possible enhancement to the C•RAM controller: the addition of a microcoded sequencer. Using a block of microcode downloaded from the host, this sequencer can generate a stream of SIMD instructions. This type of operation is similar to some previous SIMD controllers.

In order for the inclusion of a microcoded sequencer to offer an advantage, the amount of time required by the host to configure the sequencer must be significantly less than the time required by the host to issue the SIMD instructions directly. This is achieved by implementing the sequencer with support for loops and taking advantage of the repetitive nature of SIMD instructions in functions. For "long running" SIMD functions, the host can off-load the task of issuing SIMD instructions to the controller-based sequencer. A paper design for one possible implementation of a sequencer on the controller is included in appendix B.

The controller–based sequencer does not allow the host to access the C•RAM device directly. As a result, the controller–based sequencer is intended to co-exist with the controller used by the host sequencer. With a combination of the two, the host can sequence short SIMD instruction streams and access the DRAM memory core, as well as use the controller–based sequencer for sequencing functions of SIMD instructions.

The use of a microcoded sequencer allows the overlap of host and SIMD operations, but whether there is much opportunity for this in applications has yet to be investigated. Unfortunately, the addition of a sequencer increases the complexity of the controller significantly, something which might not be desirable in certain implementations of the C•RAM controller.

3.4 Summary

In this chapter we presented an architecture for a C•RAM controller that assigns the role of sequencer to the host microprocessor. This simplifies the controller, which in the process reduces the latency that it introduces between the host and the SIMD array when compared to a controller with a microcoded sequencer. Provided with a low-latency communication path to the SIMD array, a host-based sequencer can use the flexibility and performance of a microprocessor architecture while issuing SIMD instructions.

As many C•RAM devices are expected to be DRAM-based, these devices will be affected by the latency gap between DRAM devices and a microprocessor. For systems where it is desirable to reduce the idle time that a microprocessor experiences when interacting with the C•RAM device, we have presented two enhancements to the controller. The first is a buffering mechanism that can be inserted in the SIMD instruction path to enable burst issues by the microprocessor host. The second is the addition of a microcoded sequencer that relieves the host from sequencing “long running” SIMD functions. In addition to the complexity that both these enhancements add to the controller, it is possible that the context switching of the microprocessor will nullify any benefit gained from these enhancements. As such, these enhancements are only recommended for systems that experience a recognizable benefit from the implementation of one or the other of the enhancements.

Chapter 4

System Implementation

4.1 Overview

The Embedded SIMD Machine (ESM) is an implementation of a host–sequencer C•RAM controller in an embedded system. The ESM, composed of a MSA2000 microprocessor evaluation board with an expansion board attachment, prototypes an architecture that integrates C•RAM into a system. The microprocessor component on the MSA2000, a Mitsubishi M32R/D, is used as the host for the C•RAM architecture. The accompanying C•RAM controller is implemented in a Field Programmable Gate Array (FPGA), which with the C•RAM device, an Accelerix AX256 graphics accelerator, is housed on the expansion board that is connected to the MSA2000.

The implementation of the ESM is discussed in this chapter. Section 4.2 provides an overview of the MSA2000 evaluation board. Section 4.3 describes the interface to the Accelerix AX256 that is used by the C•RAM controller. Section 4.4 covers the C•RAM daughter-board, which is the expansion board that houses the FPGA and AX256 device. Section 4.5 provides an overview of the C•RAM controller for the ESM system. Section 4.6 describes the interface between the controller and the M32R/D host. Section 4.7 provides a detailed description of the components of the controller. Section 4.8 presents the results from the synthesis of the controller.

4.2 MSA2000 Evaluation Board

The MSA2000 is an evaluation board developed by Mitsubishi for the 32-bit M32R/D RISC microprocessor. In addition to the M32R/D microprocessor the evaluation board has the following features: 8 MB of DRAM; an I/O ASIC to provide timers, a chip-select controller and other functions; flash memory for storing boot code; and two PCMCIA slots, one of which is used to hold a LAN card for Ethernet support. The MSA2000 supports remote source-level debugging of applications over Ethernet through its lan card, which eases the development of software. The board can be expanded though its multiple connectors, including two DIN96 bus connectors used by the C•RAM daughter-board. An on-board oscillator provides a 16.67 MHz clock signal for use by the microprocessor board, and any attached expansion boards.

The external interface of the M32R/D operates at 16.67 MHz, however, internally the M32R/D operations at 66.67 MHz. Even though it is a 32-bit microprocessor, the external data bus of the M32R/D is only 16-bits wide. The M32R/D can, however, coordinate burst transfers of data with an external device, because of the M32R/D's higher internal clock. These burst transfers are utilized by the C•RAM controller, when transferring 32-bit data words between the host and the AX256 device.

4.2.1 M32R/D Signals

The C•RAM controller connects directly to the address bus and the data bus of the M32R/D. The signals that are used by the controller to communicate with the M32R/D are described in this section.

Data Bus [0:15] These signals are the external 16-bit data bus of the M32R/D. Bit 15 of the data bus represents the least significant bit and bit 0 represents the most significant bit.

Address Bus [8:30] The M32R/D can address up to 4 GBytes with its address bits, but only 32 MBytes is externally addressable. This external space is divided into 16 MBytes of user space and 16 MBytes of I/O space. The address bits A8 to A30 are exported to the external address bus and are used with the SID signal to index the external address space. The address bit A31 is not exported, but is used instead to internally mask data that is being read.

$\overline{\textbf{BCH}}$ and $\overline{\textbf{BCL}}$: Byte Control Since the M32R/D does not export the least significant bit (LSB) of its address bits, A31, the active low signals $\overline{\text{BCH}}$ and $\overline{\text{BCL}}$ are used to validate the high and low bytes of the 16-bit word being transferred. Both signals are asserted for the transfer during a read, and A31 is used to mask the data internally. Only the valid byte signal is asserted during a write. If a 16-bit word is being transfered, then both signals are asserted.

SID : space identifier The M32R/D splits its address space into user space, which is cached, and I/O space, which is not cached. The SID signal is used to signal which half of the address space I/O space (SID = High), or user space (SID = Low) is being referenced by the address bus.

$\overline{\textbf{BS}}$: bus start The active low signal $\overline{\text{BS}}$ is used by the bus master, normally the M32R/D, to signal the beginning of a bus cycle. It is asserted at the beginning of a bus cycle, but does not remain asserted for the duration of the bus cycle.

R/$\overline{\textbf{W}}$: read/write The R/$\overline{\text{W}}$ signal is used to indicate whether the current bus cycle is a read cycle or a write cycle. It remains asserted as long as a bus cycle is in progress.

$\overline{\textbf{Burst}}$ The active low $\overline{\text{Burst}}$ signal indicates that a burst transfer is about to occur. Burst transfers are composed of multiple bus cycles, and this signal is used to simplify the decoding of the sequential addresses. Once it has been asserted and as long as it is asserted, the target is aware that all following addresses are sequential increments of the first address decoded. As a result, the target needs to decode only the first address and can assume the value of the following addresses.

$\overline{\text{DC}}$: data complete The active low signal $\overline{\text{DC}}$ is used by the target to acknowledge an access during a bus signal. This signal is asserted for up to a clock period after the target has completed the required operation.

4.3 Accelerix AX256 Interface

The external interface of the AX256 is composed of a 10-bit multiplexed address bus and a 32-bit bidirectional data bus, as well as supporting control signals. The most prominent of the control signals are the Row Address Strobe (RAS), the Column Address Strobe (CAS), and the Write Enable. As there is no handshaking between the AX256 and its controller, all control signals are driven by the controller.

The on-chip controller of the AX256 has to be disabled to allow an external controller to access the C•RAM architecture of the device. This is achieved by placing the AX256 in a special test mode. The following discussion is concerned with using the AX256 in this test mode.

4.3.1 Addressing

The AX256 inherits the method of addressing used by DRAM memory devices, because of its C•RAM architecture. The address is in two parts: a row address, which selects the row of memory cells to be enabled; and a column address, which selects a section or word within the enabled row. The multiplexed address bus of the AX256 is used to first load the row address followed by the column address. Both the column and row addresses are ten bits in width. The assertion of the RAS signal announces the presence of the row address, which is then latched on the next rising edge of the AX256's clock. Once the column address has been placed on the address bus, the CAS signal is asserted and the column address is latched on the next rising clock edge. These two control signals must continue to be asserted during an access, because both RAS and CAS enable the operation of their respective decoders.

There are timing requirements that must be satisfied for proper operation when

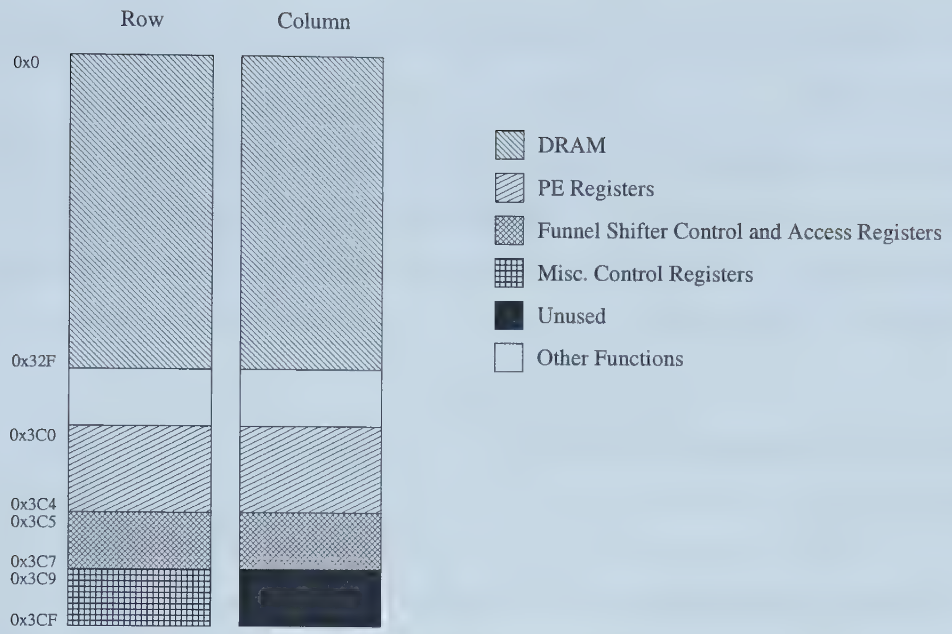

Figure 4.1: Accelerix AX256 External Interface Memory Map

addressing the AX256:

- The rated frequency of the AX256 in test mode is 25 MHz.
- A setup time of 10 ns between a valid address and the assertion of RAS is required.
- For DRAM access, a delay of 30 ns from the latching of RAS is required to allow the row enable to complete.
- A delay of 35 ns from the falling clock edge must be inserted once a row cycle has ended, in order to permit precharge before the next access.

DRAM Memory Core

The AX256 contains 13.4 Mbits of DRAM, which is divided into eight banks. Only one bank is mapped into the address space used for DRAM memory access for normal reads and writes. A control register is used to enable the banks, with the state of each bank corresponding to a bit in the register. A bank is mapped into the address space by setting the appropriate bit in the control register. The value of this control register also affects which banks of processing elements are enabled for SIMD operations.

In the situations where data needs to written into the same location in multiple banks, the necessary banks can be enabled by setting the appropriate bit in the control register. This method of multiple bank access is available only for writes and cannot be used for reads.

4.3.2 Processing Elements

Data Transfers

The processing elements in the AX256 contain five registers that can be accessed as memory. Like the DRAM memory core, the control register responsible for enabling banks also affects the registers in the PEs. Through the address space for the PE registers the host can insert operands into the PEs, without first writing them to the DRAM memory core.

The transfer of data between the DRAM core and the PE registers is performed by a write cycle to the affected DRAM row using its row address, but with a variation on the column address. The most significant bit (MSB) of the column address is used to signal that this is a transfer between a DRAM row and the PE registers; the balance of the column address is used to select the word that is transferred. The data written during this operation determines the direction of transfer. Data being transferred from a DRAM can be written to up to two PE registers, in the other direction the data transfer is one to one. The ability to transfer an entire row of data between the DRAM core and the PE registers is provided through a control register. In this case the value of the column address, except for the MSB, is ignored. The value of the bank control register affects all data transfer operations, and is used to enable any affected banks.

Data transfer between PE registers is performed by writing to a control register. The value written to the register specifies the source register and up to two destination registers. Unlike a transfer between DRAM and the PE registers, the entire row of a PE register is transfered in all enabled banks.

ALU Operations

The ALU operations are specified through three control registers. The first register is used to load the 8-bit opcode which is used by the ALU in its computation. The opcode is not actually executed at this time, but when one of the other control registers is accessed. One control register performs the execution using the MASK PE register for masking, and the other control register uses the value of the C•RAM data bus for the purposes of masking the execution of the opcode. The value of the control register for enabling banks determines which banks are participating in the computation. Anywhere from 512 PEs up to 4096 PEs can be selected for performing the opcode execution based on how many banks are enabled.

4.3.3 Funnel Shifter

The funnel shifter in the AX256 is used to perform point to point data communication between the PEs. It can be loaded from either one of the PE registers or the AX256 data bus, and it can transfer data back to either one of these locations. A control register governs its operation, by setting the direction of shifting and the count for the number of shifts to be performed. Data access to the funnel shifter from the AX256 data bus is through an access register. A read from this register retrieves the shifted result from the funnel shifter; a write loads data into the register of the funnel shifter. A control register defines usage of the funnel shifter with a PE register. The value written to this register sets the direction of transfer, specifies the portion of the funnel shifter register affected (either bits 63 to 32 or bits 31 to 0), and selects the affected PE register.

4.4 C•RAM Daughter-board

4.4.1 MSA2000 Interface

The C•RAM daughter board connects to the MSA2000 through two DIN96 connectors. These connectors provide access to the microprocessor address and data buses, chip select controller, a clock signal at 16.67 MHz, and power (3.3V and 5V).

4.4.2 FPGA

For the purposes of implementing the C•RAM controller, the daughter board has an Altera 10K100A FPGA. This device has a capacity equivalent to between 62,000 and 158,000 gates, and has 24,576 RAM bits. The size of the device makes it possible to implement a complex logic design, e.g. a microprocessor, if there is a need to operate the daughter board autonomously. The FPGA device is volatile, as it uses SRAM cells for logic configuration, and must be reconfigured if power is disconnected. A configuration ROM can be used to address the volatile nature of the FPGA and is described later.

The FPGA is connected to the two headers that make up the MSA2000 interface, and completely isolates the C•RAM device from the microprocessor. The FPGA generates all the signals used by the C•RAM device, and as a result at least a minimal amount of control logic has to be implemented in the FPGA.

4.4.3 C•RAM Device

The C•RAM device intended for use with this board is the Accelerix AX256 graphics accelerator. The AX256 is not placed directly on the board, but rather in a "clamshell" that is soldered onto the board. This allows the chip to be replaced or swapped without requiring a new board, which is quite an important feature when one considers the prototype nature of the Accelerix chips.

The Accelerix device has an on-chip controller that has to be disabled in order to access the internal C•RAM architecture. For this purpose the Accelerix device is operated in a test mode, which permits the required type of access. Not all the signals of the AX256 are used in this special mode, therefore only the signals that are active in the test mode are connected to the FPGA.

4.4.4 In System Programming

Due to the volatile nature of the FPGA and the need for different logic configurations, the board contains circuitry to support in-system programming of the FPGA. The circuitry supports two methods of programming the FPGA: using the Altera

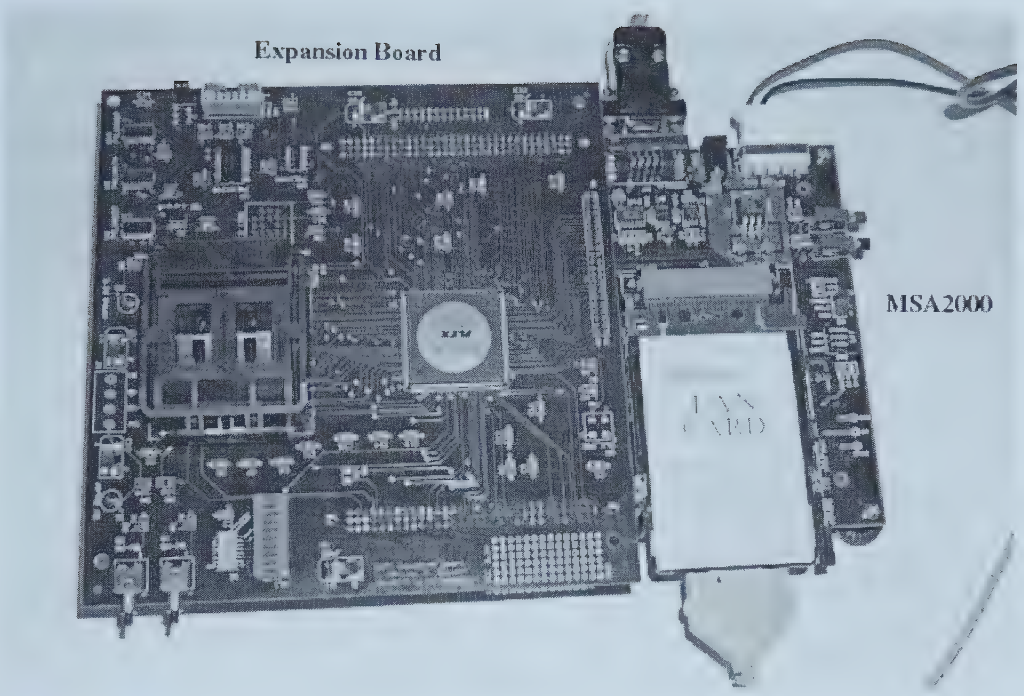

Figure 4.2: Embedded SIMD Machine System

ByteBlaster cable or using the M32R/D on the MSA2000 board, both of which use the JTAG circuitry built into the FPGA.

If the logic configuration of the FPGA is not expected to change, the configuration can be stored on-board using an Altera EPC2 configuration device. This device can be programmed using either the Altera ByteBlaster or the M32R/D, in a manner similar to the FPGA. On the power up of the board the EPC2 will program the FPGA with the stored configuration.

4.4.5 Additional Information

The technical report [31] for the C•RAM daughter-board covers its design and is the source for additional information about the board.

4.5 Overview of the ESM C•RAM Controller

The C•RAM controller used in the Embedded SIMD Machine is based on the host–sequencer scheme. Designed using VHDL, the C•RAM controller is implemented in an Altera 10K100ARC240-3 FPGA on the C•RAM daughter-board. It interfaces directly both with the data and address buses of the M32R/D, and the data and address buses of the Accelerix AX256 C•RAM device.

As the host microprocessor is responsible for sequencing SIMD instructions to the C•RAM architecture, the type of access it requires was analyzed. The results of this analysis showed that the host requires four types of access to the AX256. These four types or modes are:

- 32-bit Write by the host to the C•RAM device.
- 32-bit Read by the host from the C•RAM device.
- 8-bit Write by the host to the C•RAM device.
- 8-bit Read by the host from the C•RAM device.

Each function of the AX256 C•RAM architecture can be accessed through one or more of these modes, and the controller maps the SIMD and memory functions of the AX256 into the address space of the M32R/D. The host microprocessor accesses a function by using one of the four modes to access the address assigned to that function. For example to issue a SIMD op-code and execute it, the host first loads the 8-bit op-code by performing an 8-bit write with the op-code as data to the appropriate address, and then an 8-bit write to the address for executing the op-code. Table 4.1 lists which access modes can be used for each function.

The M32R/D transfers data in chunks of 16-bits when accessing external devices, so the two 8-bit modes that transfer 8-bit data are treated as 16-bit accesses. The M32R/D does not export its least significant address bit, A31, but instead internally masks the data. As a result the controller treats the 8-bit modes as 16-bit modes, which does not carry any penalty as the same number of clock cycles is still required to perform an access.

The C•RAM controller uses the M32R/D bus clock, at 16.67 MHz, for its system clock. This clock signal is inverted and fed to the AX256 C•RAM device as its system clock. The clock inversion allows the control signals generated by the controller to be recognized earlier in the access. If the controller and the C•RAM

Table 4.1: C•RAM Controller Access Modes

Operation	32-bit Write	32-bit Read	8-bit Write	8-bit Read
Accessing DRAM	Yes	Yes	No	No
Accessing PE Registers	Yes	Yes	No	No
Control Register for Funnel Shifter ⇔ PE register transfers	No	No	Yes	No
Control Register for accessing PE registers	No	No	Yes	Yes
Control Register for Enabling Banks	No	No	Yes	Yes
Control Register for loading the C•RAM opcode	No	No	Yes	Yes
Control Register for executing opcode with MASK PE register	No	No	Yes	No
Control Register for executing opcode with C•RAM data bus value	Yes	No	No	No
Control Register for PE register ⇒ PE register transfers	No	No	Yes	No
Control Register for Funnel Shifter operation	No	No	Yes	Yes
Access Register for accessing Funnel Shifter from the C•RAM data bus	Yes	Yes	No	No

device operated on the same clock edge, the assertion of control signals would have to be extended to satisfy setup and hold requirements.

4.6 The Host – Controller Interface

The M32R/D has a logical address that is 32-bits wide and provides a linear 4 GByte address space. This address space is divided into user space from *H'0000 0000* to *H'7FFF FFFF*, and I/O space from *H'8000 0000* to *H'FFFF FFFF*. The user space is cached and is used for the internal DRAM of the M32R/D, as well as external memory that resides in the region *H'0000 0000* to *H'00FF FFFF*. The I/O space is not cached and is intended for user, system, and internal I/O. The region in the I/O space from *H'FF00 0000* to *H'FFFF FFFF* is mapped to the external address bus.

While the C•RAM device contains a memory core. it is not desirable to cache this memory using the M32R/D caching hardware. As data in the memory core can be modified by the C•RAM PEs. maintaining cache coherence would be extremely difficult and carries too much of a penalty to be of benefit to the system operation. For this reason the C•RAM controller is mapped into the external I/O space of the M32R/D, where no caching occurs.

The C•RAM controller uses a single 512K segment of the M32R/D address space to provide access to the AX256. This segment of address space has to reside on a 512K boundary. where A13 to A31 are equal to zero. Addressing for all mapped functions of the AX256. including the DRAM memory. is relative to this boundary. Table 4.2 illustrates how the controller has allocated address space within the 512K segment to the functions of the AX256.

For most functions the address path logic in the controller has to translate the value on the M32R/D address bus to a C•RAM address composed of a row portion and a column portion. The exception is access to the DRAM memory core. The addresses used to access the memory core do not need to be converted and the valid address bits can be passed straight through. For all other functions the row address portion is generated by the address path logic from the M32R/D address. but the column address is obtained from the appropriate bits of the M32R/D address bus. Which bits of the M32R/D address bus are used is dependent on the particular function being accessed. Unused bits of address are simply ignored.

Figure 4.3 illustrates the mapping of the M32R/D address bits to C•RAM address bits. This mapping is used for any bits in the M32R/D address that are used unmodified by the controller for a function. The bits A13 to A22 of the M32R/D address map to the row address component of the C•RAM address R9 to R0. The bits A23 to A29 of the M32R/D address map to the C•RAM column address bits C9 to C3. The column address bits C2 to C0 are not used. as is explained in the appendices.

M32R/D Address Bit

A13	A14	A15	A16	A17	A18	A19	A20	A21	A22	A23	A24	A25	A26	A27	A28	A29	A30
R9	*R8*	*R7*	*R6*	*R5*	*R4*	*R3*	*R2*	*R1*	*R0*	*C9*	*C8*	*C7*	*C6*	*C5*	*C4*	*C3*	

C-RAM Address Bit

Figure 4.3: Mapping M32R/D Address Bits to C•RAM Address Bits

Table 4.2: C•RAM Controller Functions

Function	**M32R/D Address**	**Direction**
DRAM Access	Base + 0 to Base + 0x65FFC [a]	Read/Write[b]
SRC PE Register	Base + 0x66000 to Base + 0x661FC	Read/Write
DEST PE Register	Base + 0x66200 to Base + 0x663FC	Read/Write
BRUSH PE Register	Base + 0x66400 to Base + 0x665FC	Read/Write
MASK PE Register	Base + 0x66600 to Base + 0x667FC	Read/Write
ROP PE Register	Base + 0x66800 to Base + 0x669FC	Read/Write
Control Register for transfers between the Funnel Shifter and PE registers	Base + 0x66A00 to Base + 0x66A3C	Write
Control Register for accessing PE registers	Base + 0x66A40	Read/Write
Control Register for Enabling Banks	Base + 0x66A42	Read/Write
Control Register for loading the C•RAM opcode	Base + 0x66A44	Read/Write
Control Register for executing opcode with MASK PE register	Base + 0x66A46	Write
Control Register for executing opcode with C•RAM data bus value	Base + 0x66A48	Write
Control Register for transfers between PE registers	Base + 0x66A4C	Write
Control Register for Funnel Shifter operation	Base + 0x66A4E	Read/Write
Access Register for accessing Funnel Shifter from the C•RAM data bus	Base + 0x66A54 to Base + 0x66A5C	Read/Write[c]

[a] A23 of the M32R/D address bus determines if this is a transfer between DRAM and the PE Registers or a normal DRAM access

[b] A transfer between DRAM and the PE registers is a write- only operation

[c] Some addresses in this range only allow a write

4.7 Controller Components

The controller, illustrated in Figure 4.4, is composed of five components:

- Address Decoder
- Address Path Logic
- M32R $\overline{\text{BS}}$ Capture Unit
- Data Path Logic
- Refresh Logic
- State Machine

Each of these components is discussed in detail in the following sub-sections. The final section presents the synthesis results from the implementation of the controller using the Altera FPGA.

4.7.1 Address Decoder

The address decoder in the C•RAM controller matches the current value on the M32R/D address bus A8 to A30, the SID, the $\overline{\text{BCH}}$, the $\overline{\text{BCL}}$, and the R/W signals against its look up table and determines the mode of access required. If the access is valid for the C•RAM controller, one of its outputs *Read32*, *Read16*, *Write32*, or *Write16* is driven high.

The address decoder is implemented as combinational logic that is connected to the state machine.

4.7.2 Address Path Logic

The address path logic translates the value on the M32R/D address bus into a C•RAM address composed of a row address and a column address. Unlike the address decoder, the address path logic does not have to deal with the entire external address bus of the M32R/D, but rather only the portion related to the C•RAM address, A13 to A30. The arrangement of the controller functions in the M32R/D

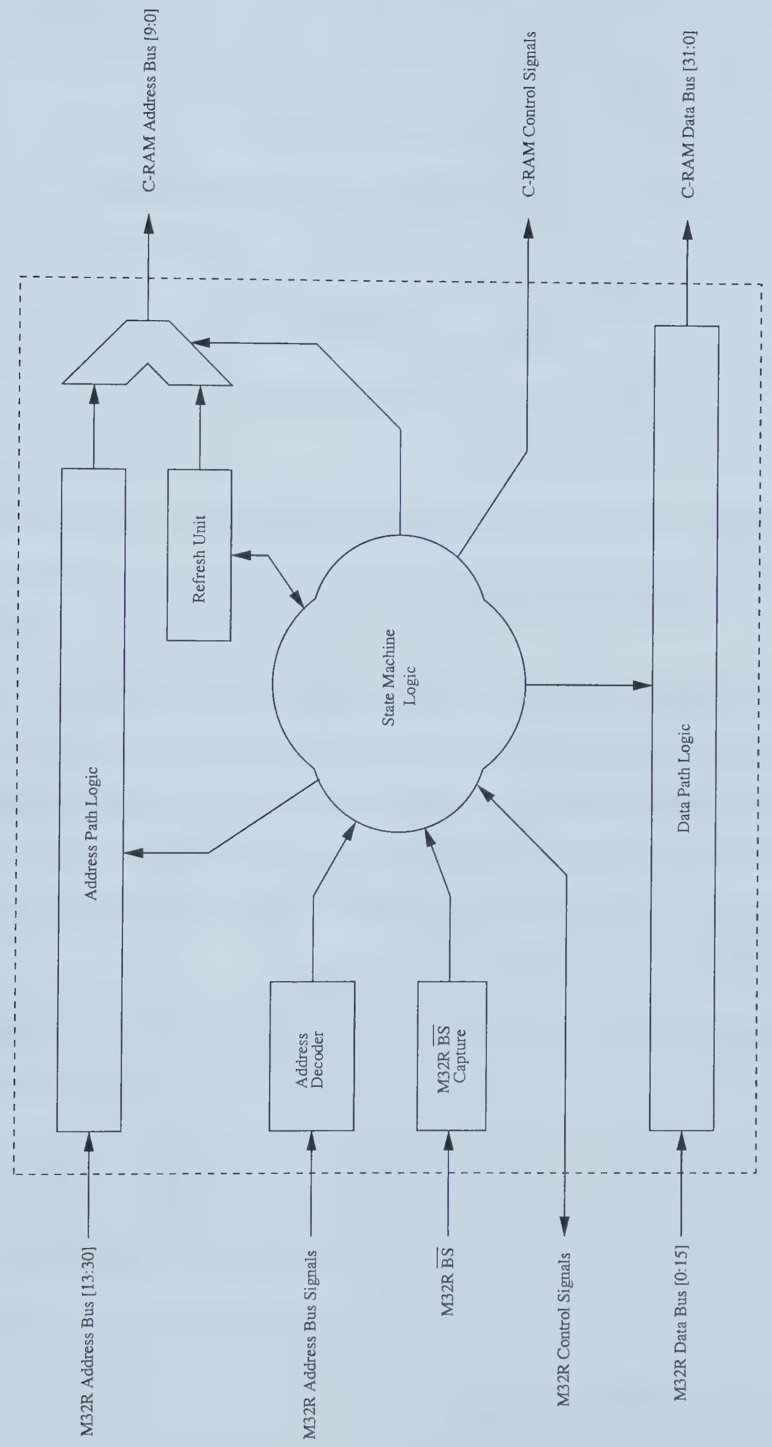

Figure 4.4: ESM C•RAM controller

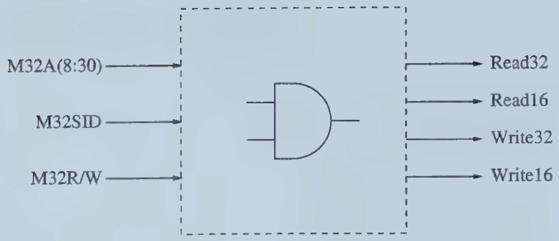

Figure 4.5: C•RAM Controller Address Decoder

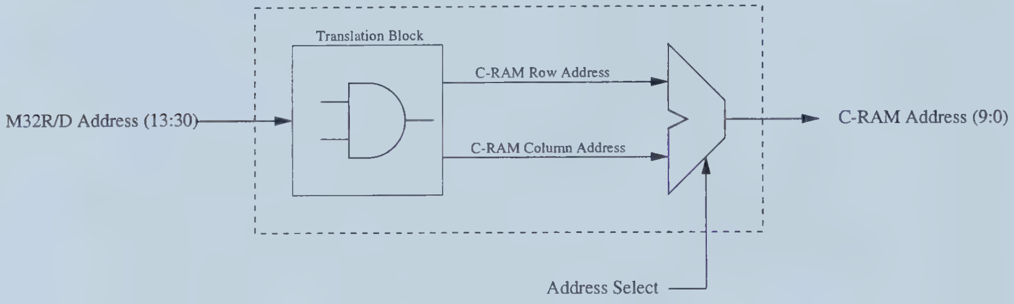

Figure 4.6: C•RAM Controller Address Path Logic

address space requires only the row portion of the C•RAM address to be generated from the M32R/D address. For the column address the appropriate bits of the address bus are passed through, if the accessed function requires a column address. As the AX256 uses a multiplexed address bus for the row and column addresses, the state machine has to switch the value being produced by the address path logic in conjunction with the appropriate control signals.

Like the address decoder, the address path logic is combinational logic and always translates the current value on the M32R/D address bus. The output by the address decoder does not have to be latched by the controller, because the value on the M32R/D address bus is constant during a bus cycle. In addition, the translated address value is only recognized by the AX256 when the appropriate control signals, i.e. RAS and CAS, are asserted.

4.7.3 M32R $\overline{\text{BS}}$ Capture

The bus start, $\overline{\text{BS}}$, signal from the M32R/D microprocessor is asserted only for approximately three quarters of a clock period after a rising clock edge. As the state machine performs its transitions on the rising clock edge, there are possible

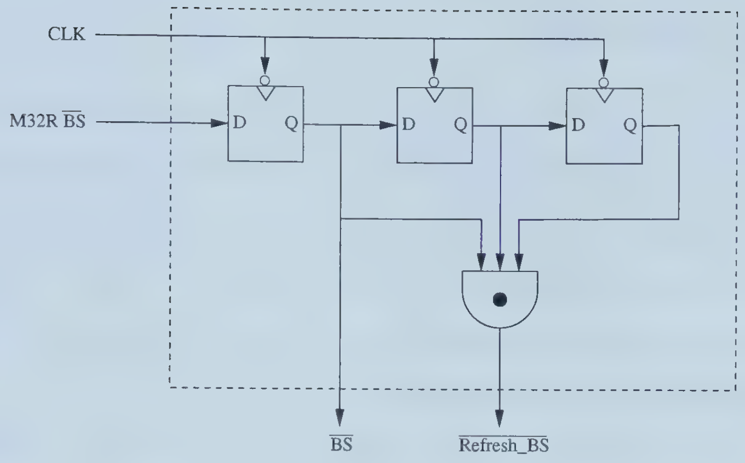

Figure 4.7: C•RAM Controller M32R $\overline{\text{BS}}$ Capture Unit

setup time violations that could cause the assertion of $\overline{\text{BS}}$ to be missed by the control logic. The capture unit addresses this by holding the asserted value of the $\overline{\text{BS}}$ signal for an additional amount of time, allowing it to be recognized.

The $\overline{\text{BS}}$ capture unit produces the two signals, $\overline{Refresh\_BS}$ and $\overline{BS}$, that represent the held values of the $\overline{\text{BS}}$ signal asserted by the M32R/D. The value of the M32R/D's $\overline{\text{BS}}$ signal is captured on the falling edge of the clock, and it is held using a set of three falling edge triggered registers. The $\overline{BS}$ output of the capture unit holds the assertion of $\overline{\text{BS}}$ for one clock period and is used by the state machine outside of a refresh cycle. The assertion of the M32R/D's $\overline{\text{BS}}$ signal is captured half a clock period after the occurrence and held until the next falling edge of the controller clock signal.

The operation of refreshing the DRAM memory core of the C•RAM device can delay the recognition of the assertion of $\overline{\text{BS}}$ by the state machine beyond the normal interval. The $\overline{Refresh\_BS}$ output holds the assertion for three clock periods, which provide sufficient time for recognition by the state machine once refresh has been completed. The $\overline{Refresh\_BS}$ is generated by using three registers and performing a logical AND of their outputs. The three registers are composed of the register used for $\overline{BS}$, and an additional two registers. All three registers are triggered by the falling edge of the controller clock; as a result the output of the capture unit should be stable by the time state progression occurs in the control logic.

4.7.4 Data Path Logic

The data path logic is used by the state machine to handle the interfacing between the data bus of the M32R/D microprocessor and that of the AX256 C•RAM device. The main issue that has to be addressed by the data path logic is the difference in bus sizes between the M32R/D's 16-bit data bus and the AX256's 32-bit data bus. Fortunately, the M32R/D is capable of burst transfers that speed up the process of performing multiple word transactions.

For access by the host in one of the 8-bit modes, the data path logic does not have to perform any action other than to pass the data through using its tristate-able I/O buffers to control the direction of data. The operation of the data path logic during an access in one of the 32-bit modes is determined by the direction. When the M32R/D is writing a 32-bit data item to the C•RAM device, the most significant word is transferred first and loaded into the MSB register. The least significant word is passed straight through and joined with the value stored in the MSB register, in order to form a 32-bit word for writing to the C•RAM device. When the M32R/D is reading a 32-bit data item, the 32-bit word is retrieved from the C•RAM device and is switched on to the M32R/D data bus in 16-bit words using a multiplexer.

4.7.5 Refresh Logic

The memory core of the AX256 consists of DRAM, which needs to be refreshed at regular intervals. The refresh logic is used to measure the time between memory refreshes, and to alert the control logic when a refresh is required. The refresh logic is composed of two parts: a timer that counts down the time left until the next refresh, and an address counter that cycles through the addresses of the DRAM rows 0 to 815.

Each row in the memory core has to be refreshed once every 16 ms. Accounting for 816 rows, the timer in the refresh logic has to issue a refresh request every $\frac{16ms}{816} = 19.6\mu s$. Requests do not have to be acknowledged right away, because they can occur while the controller is being accessed by the host. The controller has until the occurrence of the next request, i.e. 19.6 μs, to acknowledge the current refresh

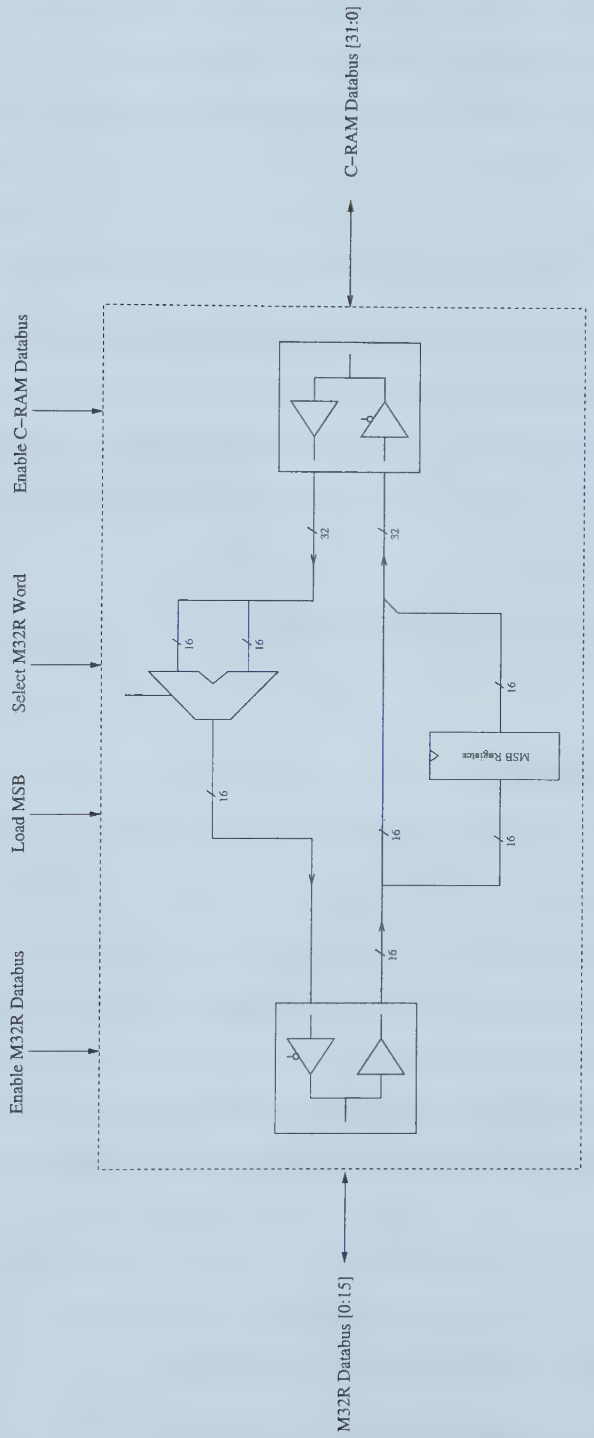

Figure 4.8: C•RAM Controller Data Path Logic

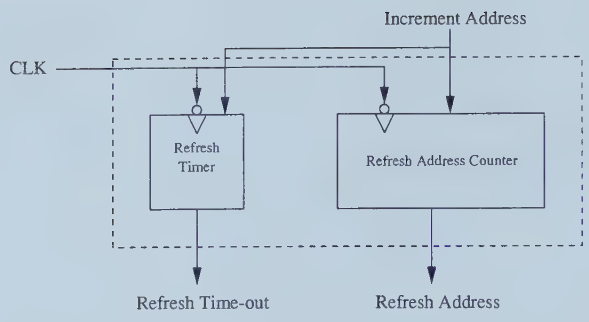

Figure 4.9: C•RAM Controller Refresh Logic

request. The state machine performs refresh by switching the value of the refresh address counter onto the C•RAM address bus and performing a row access. After a refresh is complete, the state machine increments the refresh address counter. To prevent time drift, the timer does not wait for the state machine to acknowledge a refresh request, instead it restarts and counts to the next request. The request signal, however, continues to be driven until the state machine has incremented the refresh address counter.

4.7.6 State Machine

The state machine provides the control logic for synchronizing communication with the M32R/D and the AX256. Unlike other registered logic in the controller, the state machine operates on the rising edge of the controller clock. As a result state progression occurs between the trigger edges of the other registered logic in the controller, and the AX256 device.

Previously we identified the types of low-level access to the AX256 required by the host. The operation of the state machine is divided into five branches, that include the four types of low-level access and the handling of refresh. These five branches are labeled as:

- Reads by the M32R/D with a 32-bit Data Item
- Reads by the M32R/D with an 8-bit Data Item
- Writes by the M32R/D with a 32-bit Data Item
- Writes by the M32R/D with an 8-bit Data Item

- Refreshing the memory core of the AX256 C•RAM device

The state machine resides in a default state while no operation is pending. On each rising clock edge the state machine checks its prioritized condition list, and determines whether it needs to begin state progression down a particular branch. Refreshing memory is given priority over all other operations, therefore the conditions for refresh are checked first. Refresh is the only operation that does not require interaction with the M32R/D microprocessor, as a result it is possible that a $\overline{\text{BS}}$ strobe from the M32R/D could be missed by the state machine. The $\overline{\text{BS}}$ Capture Unit is used to hold assertions of $\overline{\text{BS}}$ under normal circumstances (i.e. outside of refresh), and performs a similar task during refresh. During refresh, however, the assertion of $\overline{\text{BS}}$ has to be held for longer as the state machine will not acknowledge the assertion until it is done refreshing memory. The $\overline{\text{BS}}$ Capture Unit accommodates this by providing the $\overline{\textit{Refresh\_BS}}$ signal, which is the $\overline{\text{BS}}$ assertion held for three clock cycles. This provides sufficient time for the state machine to complete refresh and acknowledge the assertion of $\overline{\text{BS}}$.

The assignment of priority to operations in the state machine affects which operation is performed first, however, once an operation has began, i.e. state progression down a branch, it is treated as atomic and cannot be interrupted. For example, a refresh request that occurs while the host is loading an opcode, an 8-bit data write, will not be processed until the current access by the host is complete. The short interval required by each operation to complete (no more than five clock cycles) is within the tolerance of any pending operation, be it an access by the host or a refresh request. Each branch of the state machine is discussed in detail in Appendix C.

The timing diagrams Figures 4.10, 4.11, 4.12, and 4.13 illustrate the four types of low-level access used by the host. Writes by the microprocessor in both 8-bit and 32-bit modes are the operations that require the least amount of time, 3 clock cycles. Both types of writes take the same amount of time, because the controller overlaps the M32R/D bus cycle with the AX256 bus cycle and takes advantage of the M32R/D's burst mode. Reads by the host, however, are longer than writes, requiring 4 clock cycles for a read with 8-bit data and 5 clock cycles for a read with

32-bit data. The two factors that extend the length of a read beyond that of a write are: the "next clock cycle" arrival of data from the AX256, and the need to segment any 32-bit data from the AX256 into 16-bit chunks for the M32R/D bus. While the first factor affects both types of reads, the second factor only affects 32-bit reads by the host.

As previously mentioned the host uses the 8-bit and 32-bit writes for SIMD instruction issues as well as other SIMD operations, such as parallel loads of data into the PE registers. During parallel processing, access by the host will be almost exclusively writes to the controller, as a result the higher latency of reads does not impact the performance of parallel processing. The latency of reads, however, will impact the performance of the system DMA controller, if it is used with the C•RAM controller. Unfortunately the DMA controller does not have a burst signaling mechanism similar to the one used by the M32R/D, so each address would have to be validated during a transfer. From our experimental analysis it was determined that the amount of time required to repeatedly validate addresses eliminated any benefit gained from using the fast page mode of the AX256[1]. An increase in the clock frequency of the controller could be used to reduce the time required for address validation during DMA transfers, however, limited resources prevented the examination of this option.

[1]Fast page mode enables data stored in the same row of memory to be retrieved faster by keeping $\overline{\text{RAS}}$ asserted while strobing addresses with the $\overline{\text{CAS}}$

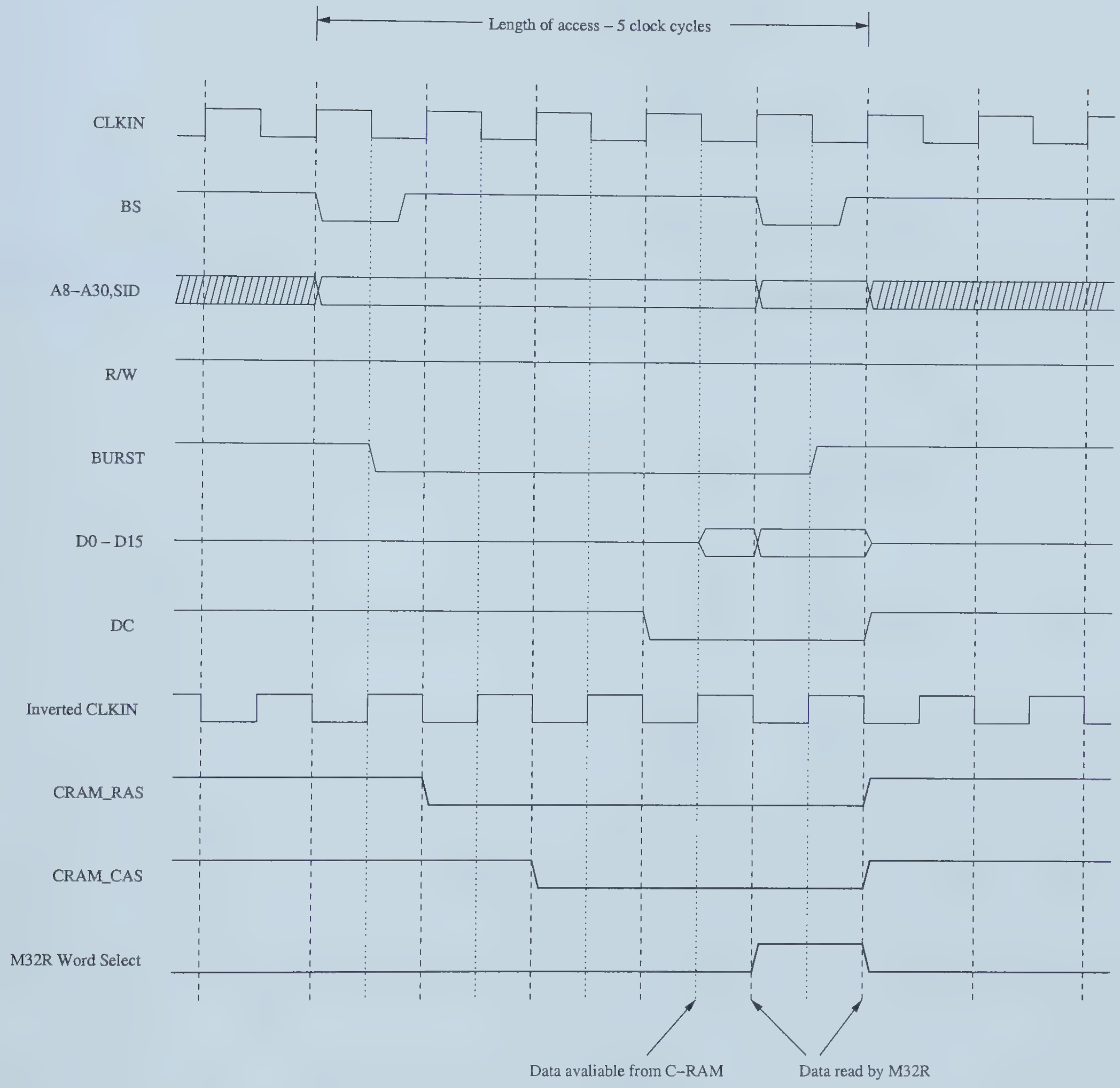

Figure 4.10: Timing Diagram of a M32R/D read with 32-bit data

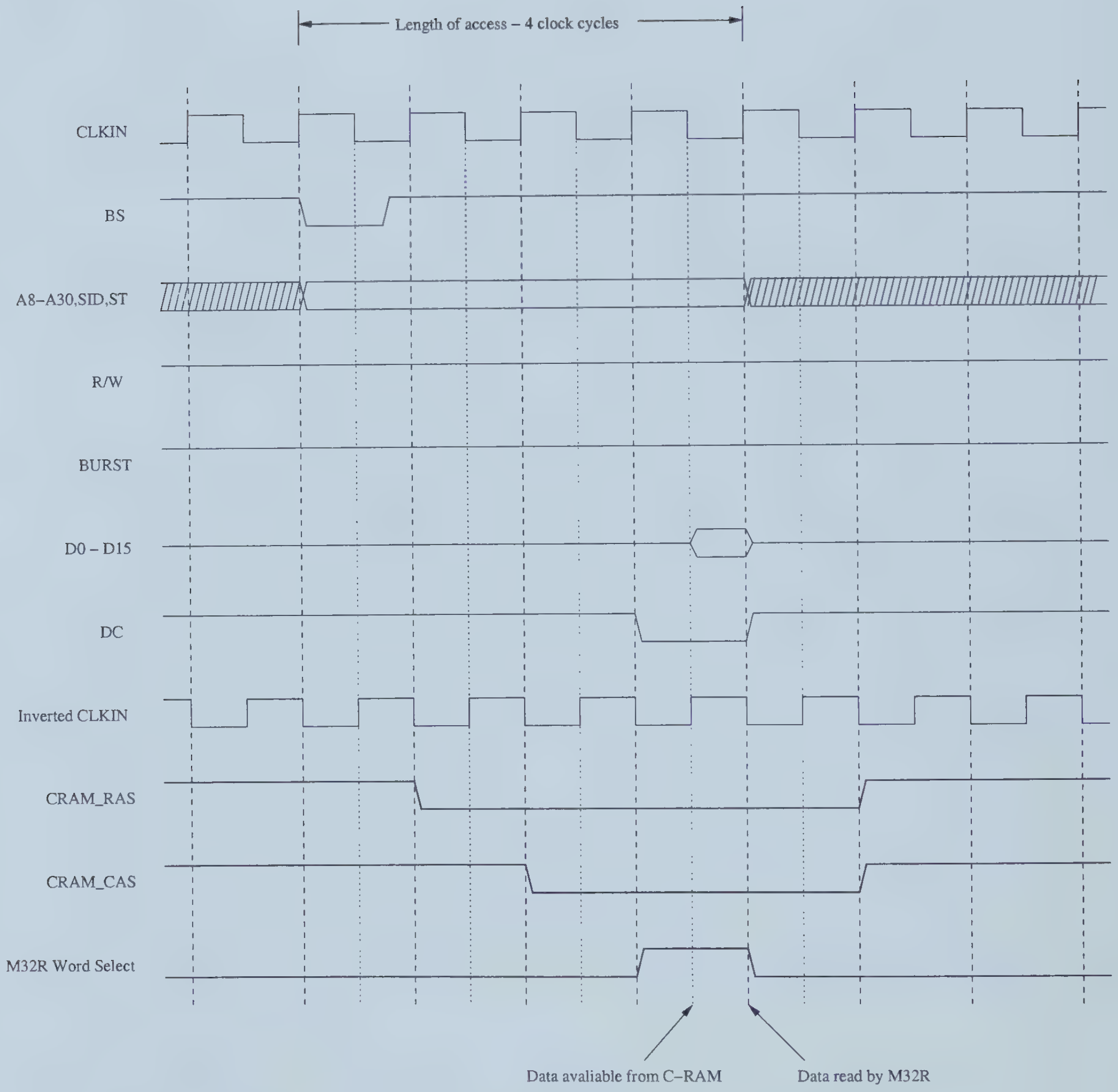

Figure 4.11: Timing Diagram of a M32R/D read with 8-bit data

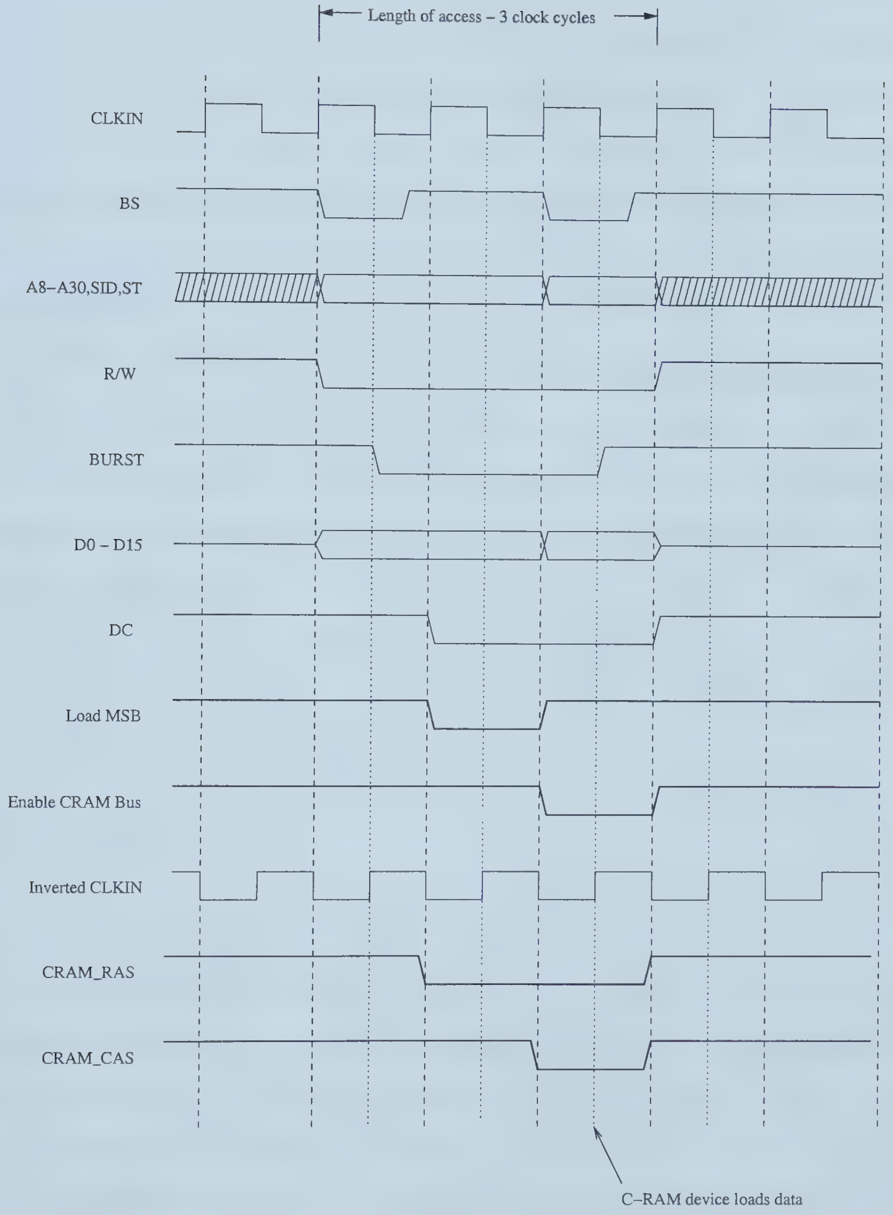

Figure 4.12: Timing Diagram of a M32R/D write with 32-bit data

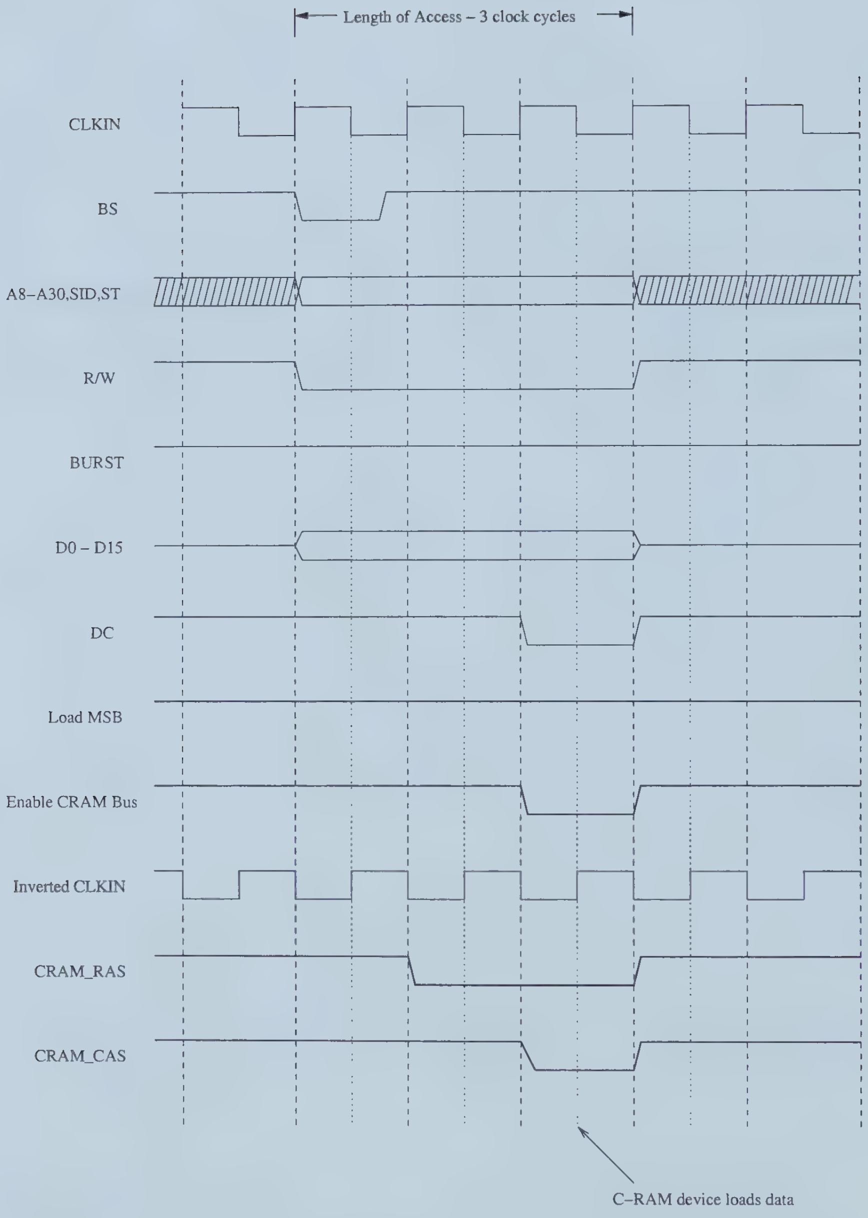

Figure 4.13: Timing Diagram of a M32R/D write with 8-bit data

4.8 Synthesis Results

4.8.1 Implementation Issues

Address Path Logic

The operation of the address path logic involves the generation of C•RAM row addresses that are based on the value of the M32R/D address bus. Two methods of performing this generation were tried: the first method was a look-up table that used the RAM bits in the 10K100ARC240-3 FPGA, and the second method was to implement the generation as combinational logic and embed the row address values within the logic.

The 10K100ARC240-3 contains RAM bits implemented in components called Embedded Array Blocks (EABs), which can be used for to implement memory. To implement the look-up table for the address path logic, a read only memory (ROM) was created to hold the values of the table. In operation the M32R/D address is decoded to an index that points to the corresponding row address in the ROM.

The implementation of the address path logic using a ROM-based look-up table, however, proved to be the slower of the two methods. As a result it was discarded for the combinational logic block implementation.

State Machine Resets

Resets, a critical part of state machines, are used to start the machine in a known state. As resets can interrupt the state progression of machines, the occurrence of glitches on the reset line is a matter for concern. The use of synchronous resets, which are acknowledged only on a clock edge, reduces a state machine's vulnerability to glitches. Unfortunately, designing a state machine with a synchronous reset cannot make use of the optimizations by the Altera tools for an asynchronous state machine. These optimizations result in reduced logic for the state machine, and allows the definition of a default start-up state through the reset description. In order to take advantage of the optimizations, the state machine for the C•RAM controller is implemented with an asynchronous reset. To minimize the occurrence of glitches, the reset is driven by an external signal connected to a push-button switch, and is not connected to the output of any combinational logic.

Propagation Delay in Data Transfer

After synthesis it was discovered that for the 10K100ARC240-3 FPGA, the propagation delay in data path logic could affect the operation of reads by the M32R/D microprocessor. The problem arises from the combination of the inversion delay of the AX256 clock signal and the propagation delay in the data path logic. The combined delay could prevent the most significant 16-bit portion of the C•RAM word from arriving at the pins of the M32R/D data bus in time to be read by the microprocessor. Figure 4.14 illustrates the effect of this delay with a 32-bit read by the M32R/D; the 16-bit read operation is affected in a similar manner.

The solution implemented to address this problem is the insertion of a wait state in the branches of the state machine that handle reads by the M32R/D. The state **s2wait_32r** is added to the branch for the 32-bit read operation, continuing the signal values of the state **s2_32r** for an additional clock period. The assertion of $\overline{\text{DC}}$ signal in the state **s3_32r** is delayed by one clock period, because of the additional state in the branch. This provides enough time for the most significant portion of the C•RAM word to propagate through the data path logic and reach the pins of the M32R/D data bus. In a similar manner, the state **s2wait_16r** is inserted into the branch for the 16-bit read operation.

If the controller is implemented in a faster FPGA, either of a higher speed grade or a different family, these wait states might not be needed. In which case they can be removed from the state machine.

4.8.2 Synthesis Summary

The synthesis of the VHDL code for the C•RAM controller resulted in an implementation that used 393 logic cells out of a possible 4992 on the 10K100ARC240-3 FPGA, which is approximately 7% of the chip capacity. Based on static timing analysis, the critical path of the C•RAM controller is identified as the address path logic, which has a worst case propagation delay of 46 ns. This is acceptable as it is less than the clock period of 60 ns and the output would be stable during state

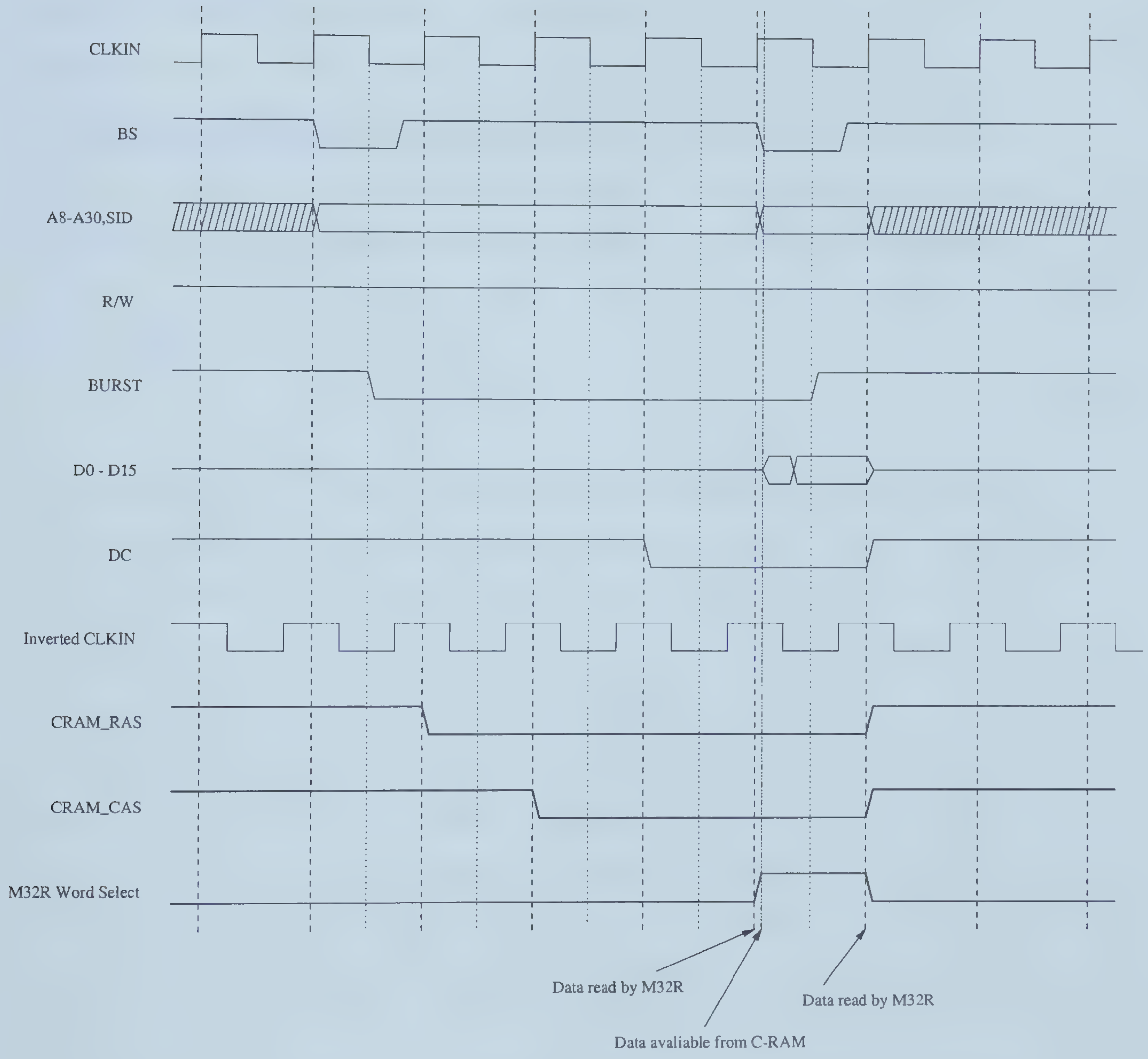

Figure 4.14: Timing Diagram of the data propagation delay in a 32-bit read

transition. With the same timing analysis, the registered logic component of the design was determined to have a maximum operating frequency of 36.23 MHz. This exceeds the requirements for the current frequency of the controller clock, and provides enough margin for significantly increasing the frequency of the system clock, if one so desired.

The clock signal used by the AX256 is generated by the controller through inversion of the controller clock. The delay introduced by this inversion is determined to be 15 ns, which is within the tolerance range for write operations and refresh, but requires wait states for reads. With the addition of the wait states for reads, the delay in the inverted clock of the AX256 is within the tolerance range for all operations.

Chapter 5

System Evaluation

5.1 Overview

The Embedded SIMD Machine is a hardware platform that is designed to integrate an Accelerix AX256 C•RAM device into a embedded system. In the absence of a functional AX256 chip the C•RAM controller in the ESM uses the deterministic timing of the AX256 to emulate its presence. This emulation of the AX256 makes it possible to evaluate the performance of the ESM and compare it to other systems. This chapter evaluates the performance of the ESM using mathematical kernels and compares it to an MSA2000 embedded system, as well as a Sun Ultra 10 workstation. The results of this comparison is presented in section 5.2. Since we are attempting to evaluate the controller and not the PE architecture, or perform feats of parallel programming, simple embarrassingly-parallel mathematical kernels will suffice as bench marks.

In Section 5.3 we use the performance of the ESM as the basis for analyzing the impact of increasing the number of C•RAM PEs that are available for computation. The kernels used to evaluate the ESM are extended to measure the improvement in performance as the number of PEs increase. An Ultra 10 acts as a control against which the performance of the PEs is compared. A summary of the results from this analysis is provided in Table 5.1.

Table 5.1: Performance of A C•RAM Equipped Workstation on Large Data Sets with Arrays of 25 x 128K Elements

Kernel	Sun Ultra 10	C•RAM Workstation (128K PEs)	Speedup
Vector Addition	371.6 ms	1.4 ms	265
Vector Multiplication	668.6 ms	24.8 ms	27
Parallel Multiplication	788.9 ms	23.4 ms	34

5.2 Evaluation Results

The evaluation of the ESM involves the use of three kernels to compare its performance to other systems. A Sun Ultra 10 and a MSA2000 are used as the control group against which the ESM is compared. The Sun Ultra 10 is a UNIX workstation, that is equipped with a UltraSPARC-IIi 333MHz microprocessor and 512 MBytes of RAM. The MSA2000 is an M32R/D microprocessor evaluation board, similar to the one used in the ESM, with a M3200D4AFP 66MHz microprocessor with 2 MBytes of RAM on-chip and 8 MBytes of RAM on the system board. As the ESM is an embedded system, the inclusion of the MSA2000 as a control provides a comparison for embedded environments. The Ultra 10 runs the Solaris operating system, which provides resource monitoring functions, such as *getrusage*, that can be used to measure the execution time of the kernels. Each kernel is run in the single user mode of the operating system and repeated multiple times to reduce the error from the granularity of the time measurement. The total time for execution is divided by the number of iterations to produce the runtime for each iteration. The ESM and the MSA2000 do not, however, use operating systems, instead a timer interrupt is used to track the execution time of the kernels.

The three kernels used for evaluating the performance of the ESM are vector addition, vector multiplication, and parallel multiplication. These are data-parallel operations, and are intended to measure the performance improvements that can be realized using C•RAM. The data sets for the kernels are the same size for all platforms, and are sized so that they would utilize all the PEs and most of the memory in a single AX256 device.

In representing the variables that are described in the kernels, a specific notation is used that takes into account the bit-serial nature of the C•RAM PEs. Bit variables are represented using pointers named in lower-case characters, e.g. *a*. The bit pointers are dereferenced using C style notation to access the value of a bit, e.g. **a*. Variables that are element size, in this case a 32-bit integer, are named using upper-case characters, but unlike bit variables use a C style array notation, e.g. *A[2][3]*, to index into an array of elements. This notation is used to illustrate whether computation occurs on a bit-level or an element-level.

In the following sub-sections each of the kernels is discussed and the results of the execution are presented.

5.2.1 Vector Addition

The vector addition kernel adds two 2-D arrays, *A* and *X*, to produce a third array, *B*, holding the result of the summation.

$$\begin{bmatrix} A_{1,1} & \cdots & A_{1,4096} \\ \vdots & \ddots & \vdots \\ A_{25,1} & \cdots & A_{25,4096} \end{bmatrix} + \begin{bmatrix} X_{1,1} & \cdots & X_{1,4096} \\ \vdots & \ddots & \vdots \\ X_{25,1} & \cdots & X_{25,4096} \end{bmatrix} = \begin{bmatrix} B_{1,1} & \cdots & B_{1,4096} \\ \vdots & \ddots & \vdots \\ B_{25,1} & \cdots & B_{25,4096} \end{bmatrix}$$

$$\text{where } B_{i,j} = A_{i,j} + X_{i,j}$$

Each array is 25 x 4096 elements in size, and each element is a 32-bit integer. This array size is determined by the memory within a single AX256. As each memory column in the AX256 has 816 bits, twenty-five 32-bit integers can fit in a column.

Uniprocessor

The pseudo-code[1] for the uniprocessor vector addition kernel is:

```
for i = 1 to 25
    for j = 1 to 4096
        B[i][j] = A[i][j] + X[i][j];
    end for
end for
```

The program code for the MSA2000 and the Ultra 10 is listed in the appendices.

[1]The orientation of the arrays has been tested and found to be optimal in all applications for C coding, but might require transposition for other programming languages.

C•RAM

The C•RAM implementation of this kernel has to take into account the bit-serial nature of the PEs. To implement addition using a C•RAM PE, a full addition (i.e. including carry) is performed for each bit in the elements. Each bit addition is done in two steps. The first step is the sum operation based on the logic equation:

$$b = a \oplus x \oplus carry$$

The second step is the computation of the carry for the next addition:

$$carry' = a \bullet x + a \bullet carry + x \bullet carry$$

At the least significant bit of the elements, the carry is reset to zero before addition is performed. Thereafter addition on each set of bits is performed using the carry from the previous addition.

It takes a PE sixty-four instructions (i.e. thirty-two summations and thirty-two carries) to perform 32-bit integer addition. The time for execution, however, is amortized across 4096 PEs performing addition in parallel.

Each array is stored in a segment within the DRAM memory core. A column from each array is assigned to each of the PEs, as illustrated in Figure 5.1. For example, *A[1-25][1]*, *B[1-25][1]*, and *X[1-25][1]* are assigned to PE 1. This layout of data does not require communication between PEs.

The implementation of the kernel for the C•RAM architecture in pseudo-code is:

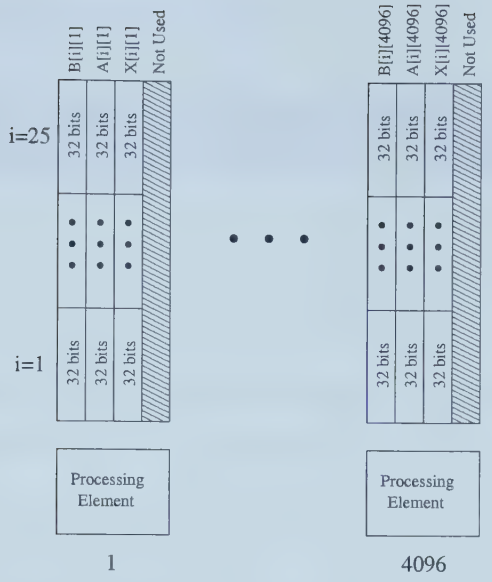

Figure 5.1: Data Layout for Vector Addition with C•RAM

```
/* Initialize Bit pointers */
initialize a,x,b;
/* Begin variable level loop */
for i = 1 to 25
    clear carry register;
    /* Begin bit level loop */
    for j = 1 to 32
        load *a into PE register;
        load *x into PE register;
        load sum opcode;
        perform operation;
        save result into *b;
        load carry opcode;
        perform operation;
        save carry in PE register;
        increment a,x,b;
    end for
end for
```

The program code for the parallel algorithm implemented on the ESM is provided in appendices.

Results

In the vector addition kernel, the ESM sees a 5x speed-up over the Ultra 10 workstation and a 75x speed-up over the MSA2000. The results are illustrated in Figure 5.2. The ESM achieves a PE utilization of 65% during the execution of this kernel. (An ideal controller that maintains 100% PE utilization would take 0.92 ms to

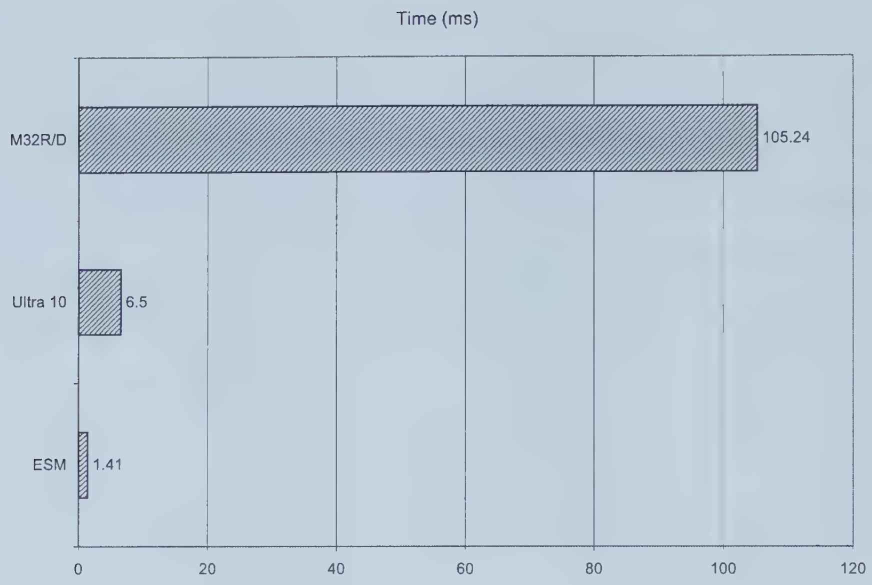

Figure 5.2: Vector Addition on the ESM using arrays of 25 x 4096 elements

execute this kernel.)

5.2.2 Parallel Multiplication

The parallel multiplication kernel multiplies the individual elements of the arrays A and X to produce the array B. Each element of B is generated according to the equation:

$$B_{i,j} = A_{i,j} \bullet X_{i,j}$$

Like the vector addition benchmark, the arrays A, X and B are 25 x 4096 elements in size, and each element is a 32-bit integer.

Uniprocessor

The uniprocessor algorithm for parallel multiplication is similar to the one for vector addition. The algorithm used is:

```
for i = 1 to 25
    for j = 1 to 4096
        B[i][j] = A[i][j] * X[i][j];
    end for
end for
```

The program code for the MSA2000 and the Ultra 10 is included in the appendices.

C•RAM

The ESM version of the parallel multiplication kernel uses an accumulate method to perform multiplication. Each PE performs multiplication for its columns of data using a data arrangement that is similar to the one used for vector addition.

The accumulate method of multiplication is based on the use of binary shifting to represent multiplication by 2. Indexing through the bits of X each time a ‘1’ is encountered the value of A is shifted to the current bit position in X and added to B. The accumulated value in B at the end of the iterations represents the result of the multiplication.

When implemented on the ESM the bits of X are used to mask the PEs, so that summation is performed for only those bit positions in X that are not zero. The shifting of A is performed by beginning summation in B from the equivalent bit position in X. As zero is shifted in for the value of A, summation below the current bit position in X is not required. The multiplication of 32-bit integers using this method is illustrated in the following pseudo-code:

```
initialize a,x,b;
for j = 1 to 32
    tmp_a = a;
    tmp_b = b;
    if *x = 1 then
        for k = j to 32
            *tmp_b = *tmp_a + *tmp_b;
            increment tmp_a;
            increment tmp_b;
        end for
    end if
    increment b;
    increment x;
end for
```

Expanding the pseudo-code to include SIMD instructions, it becomes:

```
initialize a,x,b;
for j = 1 to 32
    tmp_a = a;
    tmp_b = b;
    load *x into mask register;
    clear carry register;
    for k = j to 32
        load *tmp_a into PE register;
        load *tmp_b into PE register;
        load sum opcode;
        perform operation;
        save result to *tmp_b;
        load carry opcode;
        perform operation;
        save carry;
        increment tmp_b;
        increment tmp_a;
    end for
    increment b;
    increment x;
end for
```

Representing the multiplication as a high level operation, the pseudo-code for the benchmark can be simplified to:

```
initialize a,x,b;
for i = 1 to 25
    B[i] = A[i] * X[i];
end for
```

The program code for the implementation of parallel multiplication on the ESM is provided in the appendices.

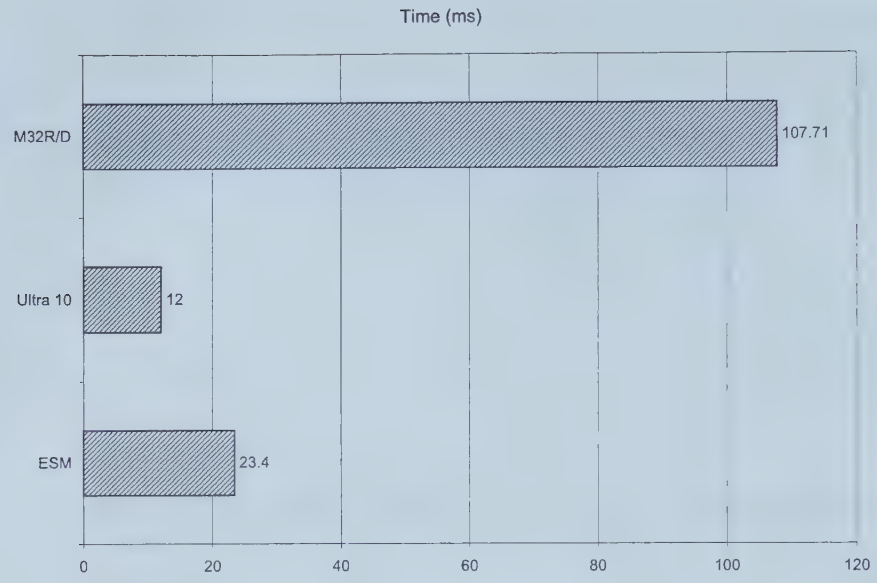

Figure 5.3: Parallel Multiplication on the ESM using arrays of 25 x 4096 elements

Results

In the parallel multiplication kernel, the performance of the ESM is worse than that of the Ultra 10 by a factor of 2, but sees a speed-up of 5x over the performance of the MSA2000. The results are illustrated in Figure 5.3. During the execution of this kernel, the ESM achieves a PE utilization of 65%. (The ideal controller with a 100% PE utilization would require 15.3 ms to execute this kernel.)

5.2.3 Vector Multiplication

The vector multiplication kernel multiplies the array A, of size 4096x25, by the vector X, of size 25x1, to produce the vector B, of size 4096x1. In algebraic terms vector multiplication is:

$$\begin{bmatrix} A_{1,1} & \cdots & A_{1,25} \\ \vdots & \ddots & \vdots \\ A_{4096,1} & \cdots & A_{4096,25} \end{bmatrix} \times \begin{bmatrix} X_1 \\ \vdots \\ X_{25} \end{bmatrix} = \begin{bmatrix} B_1 \\ \vdots \\ B_{4096} \end{bmatrix}$$

$$\text{where } B_i = \sum_{j=1}^{25} A_{i,j} \bullet X_j$$

Uniprocessor

The uniprocessor implementation is based on the following pseudo-code:

```
for i = 1 to 25
    for j = 1 to 4096 step 1
        B[j] = B[j] + A[i][j]*x[i];
    end for
end for
```

The program code for both the MSA2000 implementation and the Ultra 10 implementation is provided in the appendices.

C•RAM

The ESM implementation of vector multiplication uses a combination of parallel multiplication and vector addition. To facilitate the operation a copy of X is stored in the local memory of each PE. In the first part of the algorithm each PE performs parallel multiplication between its copy of X and its column of A. The next step involves the summation of this intermediate result to produce the value of B for each PE. The data layout for vector multiplication is illustrated in Figure 5.4. Representing addition and multiplication as high level operations, the pseudo-code for the implementation of this kernel on the ESM is:

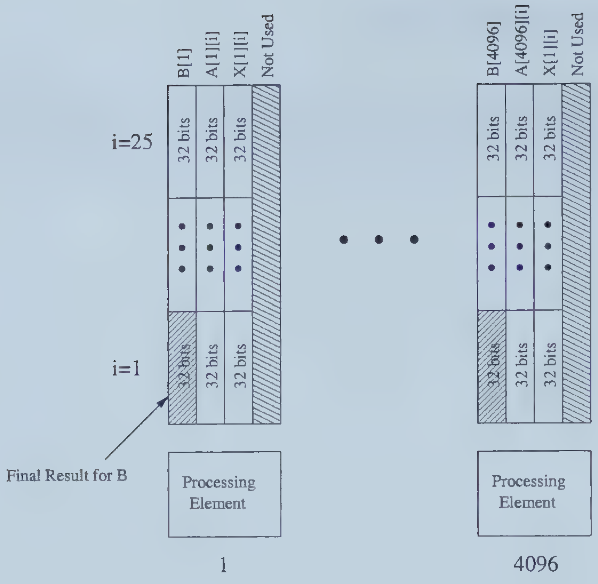

Figure 5.4: Data Layout for Vector Multiplication with C•RAM

```
initialize A,X,B;

/* Perform parallel multiplication */

for i = 1 to 25
    B[i] = A[i] * X[i];
end for

/* Perform vector Addition */

for i = 2 to 25
    B[1] = B[1] + B[i];
end for
```

The program code for the implementation of vector multiplication on ESM is provided in the appendices.

Results

In the vector multiplication kernel the ESM is slower than the Ultra 10 by a factor of 2.4, but was faster than the MSA2000 by a 3x speed-up. The results of the benchmark are illustrated in Figure 5.5. The ESM achieved a PE utilization of 65% during the execution of this kernel. (The ideal controller with a 100% PE utilization would require 16.2 ms to execute this kernel.)

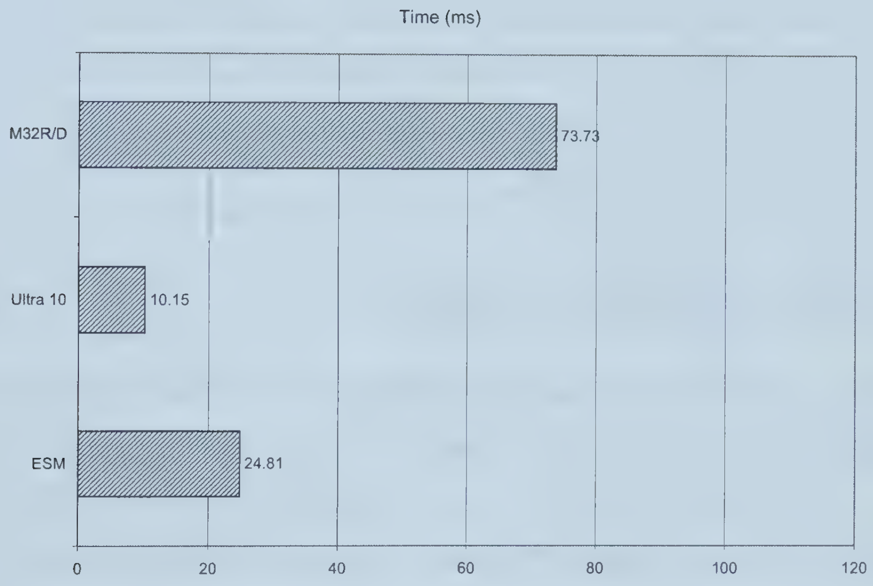

Figure 5.5: Vector Multiplication on the ESM using arrays of 25 x 4096 elements

5.2.4 Summary

The benchmark evaluation of the ESM demonstrated the performance improvements that the addition of a C•RAM architecture could bring to an embedded environment. It was also demonstrated that the proposed controller architecture, which uses the host as the sequencer, can achieve high PE utilization. When compared to a workstation, the ESM was able to offer better performance for the vector addition kernel, but lagged in performance for the other two kernels. As the amount of computation required of the C•RAM PEs increases, the performance improvements decrease. As the C•RAM PEs are bit-serial, operations that are atomic on microprocessors, such as integer addition, require a number of instruction cycles to complete on a PE. This can be addressed by increasing the number of PEs available within a system architecture. In the following section the effect of increasing the number of PEs in a system architecture is discussed.

5.3 Performance by Parallelism

In a SIMD system with simple processors, the reduced performance of less complex processing elements is offset by the number of PEs that are implemented in the system. For a C•RAM architecture to offer performance improvements, a large number of processors is required. The desired level of performance determines the number of processors that should be implemented.

With the kernels used in the previous section, a single AX256 device provided enough PEs to allow the ESM to outperform a MSA2000 embedded system. When compared to an Ultra 10 workstation, however, the ESM exceeded the performance of the Ultra 10 only in vector addition and lagged in performance for the other kernels. For a C•RAM architecture to compete against a workstation such as the Ultra 10, more PEs than were available in the ESM are required.

This section analyzes the impact of increasing the number of PEs available. Using the performance of the ESM as a measure, it is possible to project the performance improvement that a system will realize as the number of processors increase. This analysis is based on the number of PEs that would be available if C•RAM were used to replace ordinary DRAM memory, as C•RAM is designed to be a replacement for DRAM within a system. Examining an Ultra 10 workstation as an example, the system used as one of the controls in the previous section is equipped with 512 MBytes of DRAM memory. Based on the ratio used in the Accelerix AX256, replacing part or all of this with C•RAM provides access to $\sim$ 160K PEs. In the area of scientific computing, which is one of the markets for the Ultra 10, a massively parallel platform such as the one just described can be used for applications such as neural networks and image processing.

As the number of PEs implemented increases, it is expected that multiple C•RAM chips will have to be used. A couple of issues arise in a multi-chip environment: the first is inter-chip communication, and the second is signal skew on system buses. Fortunately, both of these issues have already been addressed. In the generic C•RAM architecture Elliott [5] describes methods of tapping the C•RAM communications network to enable inter-chip communication. The issue of signal skew has been addressed by skew cancellation and removal techniques that have been developed in the past [32] [33]. The effect of using a multi-chip C•RAM array on instruction issue and other control operations should be minimal. This assumption makes it possible to use the results of the ESM benchmarking to project the performance of a larger C•RAM array.

A larger C•RAM array can tackle large data sets with an increased degree of parallelism. For analysis the projected performance of the larger C•RAM array is compared to the performance of an Ultra 10 on the same size of data. As the number of PEs in a SIMD array increases, the size of the data set that can be tackled in the same amount of time increases proportionally. For example, performing vector addition on two arrays of size 8192x25 with 8192 PEs requires the same amount of time as two arrays of size 4096x25 with 4096 PEs. Using this method of analysis, the performance projected for the C•RAM array is compared to the performance of the Ultra 10 workstation.

The results of the analysis for vector addition, vector multiplication, and parallel multiplication are illustrated in Figures 5.6, 5.7, and 5.8 respectively[2]. The

[2]The effect of the microprocessor caches is visible as changes in slope at particular points in the

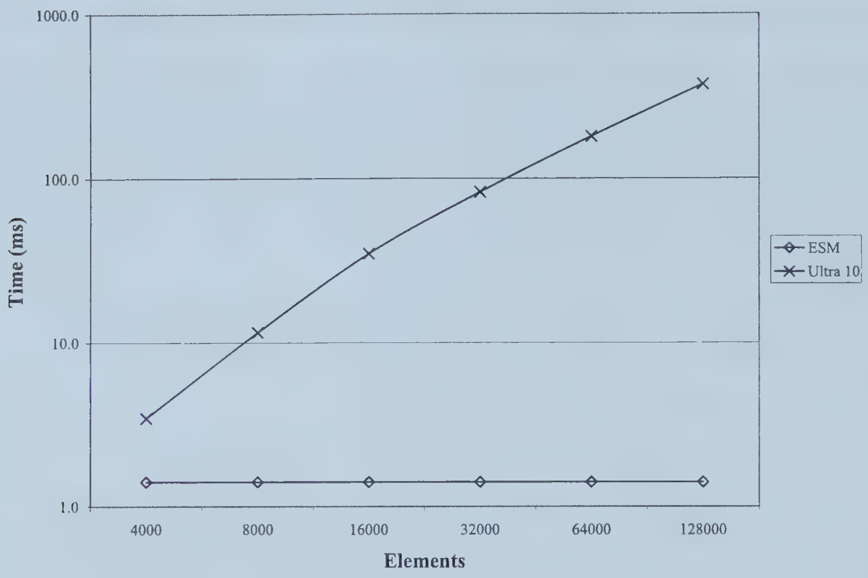

Figure 5.6: Effects of increasing parallelism: Vector Addition

largest number of PEs used for comparison is 128K PEs, and at this point a C•RAM equipped system realizes quite a significant improvement in performance. While a comparison at a lower number of PEs still shows improvements in performance, at 128K PEs the improvements in performance exceed an order of magnitude. For vector addition, the improvement in performance is over two orders of magnitude. Vector multiplication and parallel multiplication see improvements that exceed an order of magnitude.

graphs.

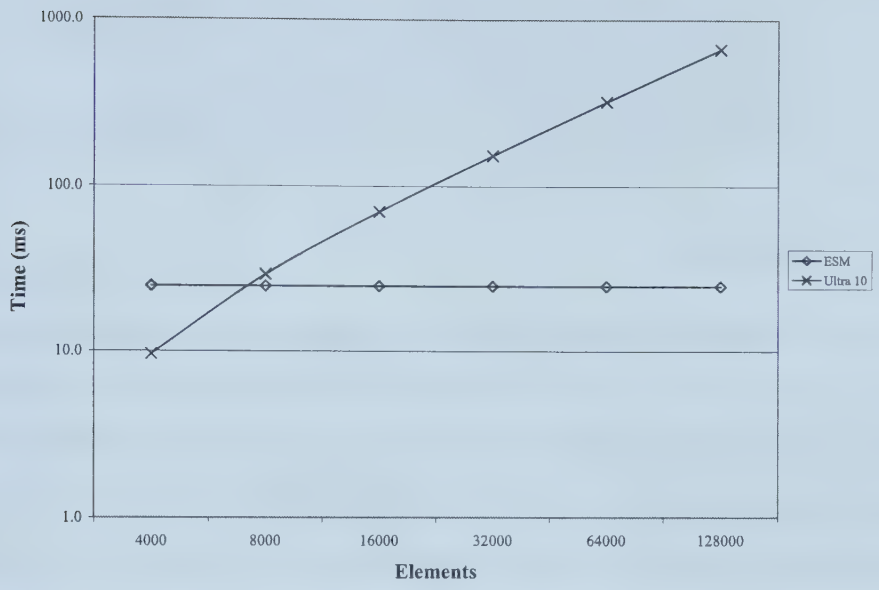

Figure 5.7: Effects of increasing parallelism: Vector Multiplication

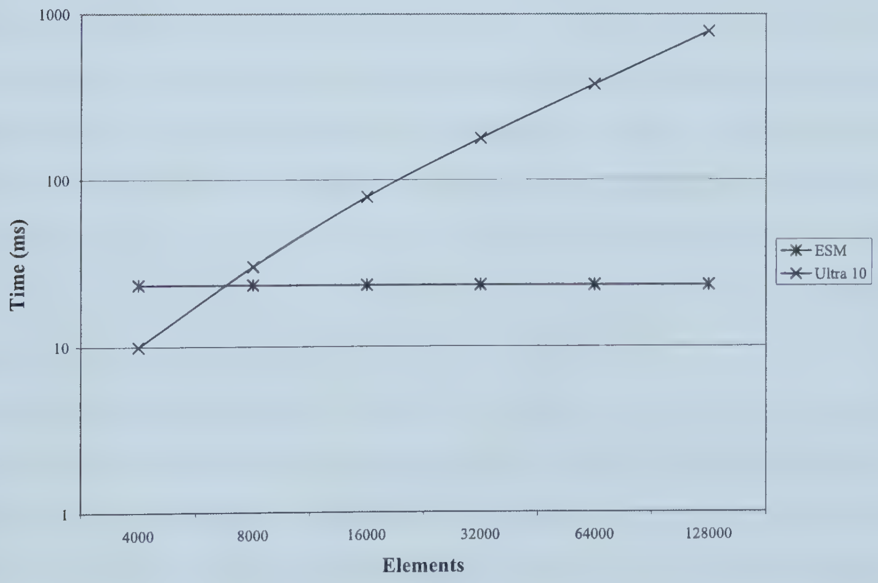

Figure 5.8: Effects of increasing parallelism: Parallel Multiplication

Table 5.2: ESM PE Utilization

Kernel	Ideal	ESM	Utilization
Vector Addition	0.95 ms	1.4 ms	65%
Vector Multiplication	16.2 ms	24.8 ms	65%
Parallel Multiplication	15.3 ms	23.4 ms	65%

5.4 Summary

The use of C•RAM in a system architecture provides an economical parallel processing platform for data–parallel applications. The performance of the ESM demonstrated that the addition of a C•RAM device to an embedded platform can provide significant speed-ups for data–parallel applications. The host-sequencer controller used in the ESM allowed the system to achieve a high PE utilization, illustrated in Table 5.2. At 65% PE utilization, the ESM demonstrated better utilization than has been reported in evaluations of Thinking Machines' CM-2 microcoded SIMD controller. The CM-2 was shown to have less than 1% utilization for a boolean parallel application designed to highlight this weakness. Using the results from the evaluation of the ESM, it was shown that systems, such as a UNIX workstation, could realize a significant performance improvement by replacing the ordinary DRAM used for system memory with C•RAM.

Chapter 6

Conclusions

In this work we have presented the results from research into integrating C•RAM devices into a system architecture. We developed a low latency C•RAM controller architecture and implemented it in a hardware platform, the Embedded SIMD Machine (ESM). From the benchmarking of the ESM, we have shown that a system equipped with a C•RAM architecture can realize improvements in performance during data-parallel processing. We demonstrated that our controller architecture could achieve a high PE utilization. C•RAM is an economical solution for parallel processing, because it can be used as ordinary memory and a massively parallel processor when required.

In Chapter 3 we addressed the integration of C•RAM into a system architecture by presenting a controller that makes a C•RAM SIMD array accessible to the host processor. We were able to simplify the design of the controller by assigning some tasks to the host processor, and taking advantage of the resources that are available as a result of improvements in microprocessor technology. Instead of using the controller to generate and sequence SIMD instructions for the array, the host processor assumes this role and the controller manages the low-level communication between the host processor and the SIMD array. The host processor issues SIMD instructions to the SIMD array through the controller, and access the memory core of the C•RAM device. By simplifying the controller we provided the host with a low latency path to the C•RAM device and made it possible to achieve a high PE utilization during data-parallel computation.

Acknowledging the increasing gap in latency between memory devices and a system processor, we examined possible enhancements to the controller that would enable the host processor to reduce its workload during the sequencing of SIMD instructions. One possible enhancement is the use of instruction buffers to enable the host to perform burst issues of instructions. These buffered instructions would then be issued to the C•RAM SIMD array by the controller, during which time the host processor is free for other system tasks. Another possible enhancement is the addition of a microcoded sequencer to the C•RAM controller similar to those used in many previous SIMD controllers. The microcoded sequencer operates by issuing SIMD instructions from a compact block of host generated microcode. A paper design for this sequencer is presented in Appendix B. The benefit of this enhancement is realized, when the time required of the host to configure the controller-based sequencer is significantly less than if the host was required to directly sequence instructions. Both these enhancements allow the overlap of parallel (on the C•RAM architecture) and sequential (on the host) processing, whether the opportunity exists in applications is yet to be investigated. In addition these enhancements increase the complexity of the controller, which may not be desirable in some implementations.

In Chapter 4 we present the Embedded SIMD Machine (ESM), a hardware platform that implements the C•RAM controller described in Chapter 3. Intended to function with the Accelerix AX256 C•RAM device, the ESM system architecture consists of a M32R/D evaluation board, the MSA2000, and an expansion board that houses the C•RAM controller and the associated C•RAM device. The controller, implemented in an FPGA, provides the M32R/D microprocessor with access to the 4096 C•RAM PEs and the 13.4 Mbits of DRAM that are available in an AX256 device.

The performance of the ESM is evaluated in Chapter 5, where it is compared to a MSA2000 embedded microprocessor system, and a Sun Ultra 10 workstation. Without a C•RAM device present in the system, the ESM controller issues correctly timed instructions, which provides an emulation for evaluation purposes. Using data-parallel kernels it is shown that the ESM performs significantly better than the uniprocessor MSA2000. Comparing the ESM to the Ultra 10 workstation using the same data-parallel kernels, the ESM performed better in only one kernel and lagged behind the Ultra 10 in the other two. The evaluation of the ESM also demonstrated the high PE utilization that could be achieved with the proposed controller architecture. The results from the evaluation of the ESM are used to analyze the impact of replacing a workstation's DRAM memory with C•RAM and increasing the number of processing elements available. It is shown that a C•RAM-equipped workstation could realize performance improvements of greater than an order of magnitude over a uniprocessor workstation, such as the Ultra 10, when executing data-parallel applications.

A C•RAM machine is capable of offering significant performance gains to a system, when enough parallelism exists in an application. In the situations where such parallelism does not exist, however, the performance improvements introduced by the addition of a C•RAM architecture are minimal or non-existent. In addition, because C•RAM is based on SIMD, its programming model is more restrictive than that of a MIMD architecture.

While C•RAM might not be usable as a parallel architecture in many applications, we believe that enough data-parallel applications exist to make integrating C•RAM into a system architecture worthwhile. The proposed controller architecture with its low overhead would allow a system to achieve a high PE utilization during data-parallel computation. Fault simulation, data mining, and signal processing are some of the applications that have been identified as benefiting from a C•RAM architecture. The design of C•RAM as a replacement for ordinary DRAM or SRAM in system memory, does not impose a performance penalty for its use in a system. When integrated into a system architecture, C•RAM can be used as ordinary memory during applications that are not data-parallel, and as a massively parallel processor during applications that can utilize its parallelism.

In conclusion integrating C•RAM into a system architecture provides an economical parallel processing platform that can be used for a range of applications to realize improved system performance.

6.1 Future Work

This work demonstrates the improvement in performance that the addition of a C•RAM architecture brings to an embedded system. One of the application areas which would greatly benefit from C•RAM is multimedia. For computationally constrained systems such as hand-held devices, e.g. media players and PDAs, using C•RAM as a platform for providing multimedia features offers great promise. The common practice is to invest in a digital signal processor that can handle the computational load, but at significant increases in cost. Instead for a lower cost, a system could be implemented using C•RAM in place of ordinary DRAM. At the moment commercial interest in C•RAM is focused on this area. From the point of view of research, the next step would be investigating the implementation of algorithms, such as JPEG and MPEG*x*, on C•RAM architectures.

In the realm of scientific computing a data-parallel architecture is of great benefit to areas such as neural networks [6]. The performance analysis of a C•RAM equipped workstation demonstrated the advantages offered to a highly parallel application. A significant hurdle, however, is the complexity of workstation environments when compared to an embedded system. Research could be conducted into developing solutions for integrating C•RAM into a workstation environment. Investigations into both software integration, such as operating system support, and hardware integration, such as cache support, could be conducted.

Bibliography

[1] Duncan G. Elliott et al. Computational RAM: Implementing Processors in Memory. *IEEE Design & Test of Computers*, 16(1):32–41, January 1999.

[2] W.A. Wulf and S.A. McKee. Hitting the Memory Wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, March 1995.

[3] R. Torrance et al. 33 GB/s 13.4 Mb integrated graphics accelerator and frame buffer. In *Proceedings of the 1998 IEEE 45th International Solid-State Circuits Conference*, pages 340–341, 1998.

[4] Albert L.C. Kwong. Parallel Fault Simulation on the C•RAM Architecture. Master's thesis, University of Alberta, 1998.

[5] Duncan G. Elliott. *Computational RAM: A Memory-SIMD Hybrid.* PhD thesis, University of Toronto, 1997.

[6] M.E. Azema-Barac. A conceptual framework for implementing neural networks on massively parallel machines. In *Proceedings of the Sixth International Parallel Processing Symposium*, pages 527–530, March 1992.

[7] Michael J. Flynn. Very High Speed Computing Systems. *Proceedings of the IEEE*, 54(12):1901–1909, Dec 1966.

[8] Dezso Sima, Terence Fountain, Peter Kacsuk. *Advanced Computer Architectures: A Design Space Approach.* Addison-Wesley, 1997.

[9] David A. Patterson and John L. Hennessy. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Francisco, CA, USA, second edition, 1996.

[10] J. P. Strong. Computations on the massively parallel processor at the Goddard Space Flight Center. *Proceedings of the IEEE*, 79(4):548–558, April 1991.

[11] Mooly Eden and Michael Kagan. Pentium processor with MMX technology. In *Proceedings of the 1997 IEEE COMPCON Conference*, pages 260–262, 1997.

[12] Stuart Oberman, Greg Favor, and Fred Weber. AMD 3DNOW! Technology: Architecture and Implementations. *IEEE Micro*, 19(2):37–48, Mar/Apr 1999.

[13] Gene Frantz. Digital Signal Processor Trends. *IEEE Micro*, 20(6):52–59, Nov/Dec 2000.

[14] W. Hinrichs et al. A 1.3-GOPS parallel DSP for high-performance image-processing applications. *IEEE Journal of Solid State Circuits*, 35(7):946–952, July 2000.

[15] Kazunari Inoue et al. A 10Mb 3D Frame Buffer Memory with Z-Compare and Alpha-Blend Units. In *Proceedings of The IEEE International Solid-State Circuits Conference*, pages 302–303, 1995.

[16] T. Simon and A.P. Chandrakasan. An ultra low power adaptive wavelet video encoder with integrated memory. *IEEE Journal of Solid-State Circuits*, 35(4):572–582, April 2000.

[17] Harold S. Stone. A Logic-in-Memory Computer. *IEEE Transactions on Computers*, C-19(1):73–78, January 1970.

[18] D. G. Elliott and W. M. Snelgrove. C•RAM: Memory with a Fast SIMD Processor. In *Proceedings of the Canadian Conference on VLSI*, pages 3.3.1–3.3.6, Ottawa, October 1990.

[19] Christian Cojocaru. Computational RAM: Implementation and Bit-Parallel Architecture. Master's thesis, Carleton University, January 1995.

[20] Dennis Fielder, James Derbyshire, Peter Gillingham, Randy Torrance, Cormac O'Connell. Single Chip Frame Buffer and Graphics Accelerator. United States Patent Number 5,694,143, December 1997.

[21] D. Jones, I. Mes, B. Hold. ASM1 Frame Buffer Functional Specification. Internal Documentation, May 1997.

[22] D. Parkinson et al. The AMT DAP 500. In *Proceedings of the 1988 IEEE COMPCON Conference*, pages 196–199, 1988.

[23] J. C. Gealow, F. P. Herrmann, L. T. Hsu, C. G. Sodini. System design for pixel-parallel image processing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 4(1):32–41, March 1996.

[24] Lewis W. Tucker and George G. Robertson. Architecture and Applications of the Connection Machine. *IEEE Computer*, 21(8):26–33, August 1988.

[25] Tom Blank. The MasPar MP-1 Architecture. In *Proceedings of the 1990 IEEE COMPCON Conference*, pages 20–24, 1990.

[26] Maya Gokhale et al. Processing in memory: the Terasys massively parallel PIM array. *IEEE Computer*, 28(8):23–31, April 1995.

[27] Accelerix Incorporated. *AX256-1M, AX256-2M, AX256-4M Ultra-High Performance True Single-Chip Flat Panel/CRT Graphics Accelerator Data Sheet*, 0.9 edition, 1998.

[28] Peter M. Nyasulu. *System Design for a Computational-RAM Logic-In-Memory Parallel-Processing Machine*. PhD thesis, Carleton University, Ottawa, Ontario, Canada, May 1999.

[29] Duncan G. Elliott. Computational RAM: A Memory-SIMD Hybrid. Technical report, Dept. of Electrical and Computer Engineering, University of Toronto, 1998.

[30] Noah Aklilu, Duncan Elliott, Curtis Wickman. A Tightly Coupled Hybrid SIMD/SISD System. In *Proceedings of The 1999 IEEE Canadian Conference on Electrical and Computer Engineering*, May 1999.

[31] Noah Aklilu. C•RAM Daugther Board for the MSA2000 Evaluation Board. Technical report, Dept. of Electrical and Computer Engineering, University of Alberta, 2001.

[32] H. Sutoh, K. Yamakoshi, M. Ino. Circuit technique for skew-free clock distribution. In *Proceedings of The IEEE Custom Integrated Circuits Conference*, pages 163–166, May 1995.

[33] Sung-Ho Wang, et. al. A 500-Mb/s quadruple data rate SDRAM interface using a skew cancellation technique. *IEEE Journal of Solid State Circuits*, 36(4):648–657, April 2001.

Appendix A

Differences between the AX-256 documentation and operation of the ESM controller

A.1 Determining the Column Address to Access The AX256

In order to access the AX256, the controller must perform a row and a column cycle. Not all operations require a column address, however, a column cycle must be performed for every operation. The operations that do require a column address are:

- DRAM access
- PE Register Access (referred to as PPU Register Addressing by Accelerix)
- Data transfers between the PE registers and DRAM
- Data transfers between the funnel shifter and the PE registers
- Using the Funnel Shift Access Register

The functionality of these operations is described in the Accelerix documentation. The mapping of M32R/D address bits to C•RAM address bits for the operations that use a column address is described in section 4.6. Other than the operations mentioned above, all other operations do not require a column address.

A.2 No Chunky Mode

The controller does not provide access to the "chunky" mode of the AX256 device. In order to save address space the address bits to access this mode are not seen by the host. It was determined that this was a feature that would not be used often and its functionality could be replicated at little cost through other means. Its removal does not restrict the host's access to the DRAM memory core or the SIMD architecture.

A.3 Masked Bit Values

For some operations the controller masks the data value written to prevent certain actions from occurring. When the PE control register (referred to as the PPU Control Register 1 by Accelerix) is accessed, bit 1 which enables chunky mode is always masked as "0". This prevents chunky mode from ever being enabled. When transferring data between the PE registers and the DRAM memory core, bit 6 of the data written is masked as "0"[1]. This makes sure that the transfer is always between the DRAM memory core and the PE registers.

[1]A value of "1" is a transfer between DRAM and the Serial Output Register

Appendix B

A Paper Design for a Controller based Microcoded Sequencer

B.1 Overview

The controller–based sequencer is intended to allow the host to off-load the sequencing of SIMD instructions to the controller. The sequencer interprets microcode that was generated by the host, and creates a stream of SIMD instructions for the C•RAM array.

For the use of the controller–based sequencer to be an advantage the amount of time required for the host to load the sequencer must be significantly less than the time the host would have spent sequencing the instructions directly. With this in mind the controller-based sequencer has to support looping to make the block of instructions compact and thus faster to transfer. An issue that raises with the use of loops is the use of variables within a loop. With C•RAM the variable that is expected to change within loops is the row address, as it is used to index into the memory columns of processing elements. The sequencer addresses this by allowing the use of both references and constant values when referring to row addresses. The references point to a bank of incrementing registers, that can be configured with variable increments to represent different step values in the row addresses.

For most C•RAM instructions, the data in the instruction is eight bits wide. A few instructions, e.g. writing to a mask register, however, require data values that are thirty-two bits in width. Rather than maintain an allocation of thirty-two bits for the data value in these instances only an eight bit field is used, and large values

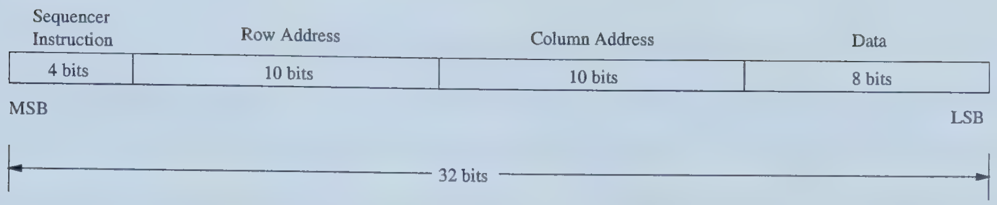

Figure B.1: Sequencer Instruction Format

can be referenced from a bank of 32-bit registers. Unlike the registers used for row addressing, these registers do not need to be incrementing registers.

The following sections discuss the architecture of the controller-based sequencer. The format of the instructions generated by the host is discussed in section B.2, followed by the design of the sequencer in section B.3.

B.2 Instructions

The instructions that the host generates for the sequencer are packed into a 32-bit word. As most high performance microprocessors are at least 32-bits, this width of a word should fit within the computational architecture of the host. Each instruction word is composed of four parts: a micro-instruction to support the operation of the sequencer (four bits), the row address component of the C•RAM instruction (ten bits), the column address component of the C•RAM instruction (ten bits), and the data associated with the C•RAM instruction (eight bits). The format of the instruction word is illustrated in Figure B.1.

Instructions to the C•RAM device are issued using write cycles, and read cycles are used only for reading data from the DRAM memory core or the PE registers. As the sequencer is issuing only instructions to the C•RAM device and is not concerned with reading data, it only needs to perform write cycles. Therefore no mechanism is required to distinguish between read and write cycles, as only write cycles will be used.

Micro-instructions for the sequencer are grouped into two functions, register referencing and loop control. Some of these micro-instructions can be packed alongside a C•RAM instruction[1] and can be performed simultaneously in a VLIW-

[1]The C•RAM instruction is stored in the row address, column address and data fields of the

Table B.1: Sequencer Micro-instructions

Micro-instruction	**Function**	**With C•RAM**
0000	Normal C•RAM instruction issue	Yes
0001	Row address in the C•RAM instruction is referenced	Yes
0010	Set row address register value	No
0011	Increment row address register(s)	No
0100	Data value in the C•RAM instruction is referenced	Yes
0101	Both the row address and data value of the C•RAM instruction are referenced	Yes
1000	Beginning of a loop	No
1001	End of a loop	No

style of operation. Table B.1 lists the sequencer micro-instructions and highlights which micro-instructions can be packed with a C•RAM instruction.

Sequencer Micro-instructions

Normal C•RAM instruction issue With this micro-instruction the host informs the sequencer that no operation is required of it except to sequence the C•RAM instruction.

Row address in the C•RAM instruction is referenced This micro-instruction alerts the sequencer that the C•RAM instruction contains a referenced row address. The value in the row address field is the index into the address register file. Before the C•RAM instruction is issued, the referenced value is retrieved and inserted into the C•RAM instruction.

Set row address register value This micro-instruction is used by the host to initialize address registers with an initial value and an increment value. This allows the host to initialize addresses before the beginning of a loop and enables the nest-

instruction word

ing of loops within the function being sequenced. The new value for the address register is stored in the row address field of the instruction and the new increment is stored in the column address field. The data field is used to index the address register being initialized and can be designed to operate in one of two ways. One approach is to use the bits of the data field to enable one or more address registers for modification, but this limits the number of address registers to the number of bits in the data field. The other approach is to use the numerical value of the data field as an index pointing to a single address register. This supports a larger number of address registers, but this micro-instruction needs to be issued multiple times to initialize a group of registers with the same set of values. Unless it is expected that multiple registers will be initialized with same value frequently, the first approach does not offer any benefit and restricts the number of registers to a lower number.

Increment row address register This micro-instruction is used to increment the address registers with the value stored as their increment. How this instruction functions will be based on the number of address registers implemented. If there are only eight or less registers, then the bits of the data field in the instruction can be used to enable the registers that are being incremented. In an implementation where more than eight registers exist, the registers can be arranged in blocks of eight to continue to facilitate the increment of multiple registers at once. As there can be up to 256 registers depending on the implementation of the *Set row address register* instruction, up to 32 block can exist. The least significant five bits of the column address field can be used to select a single block, and the data field can be used to enable multiple registers within a block. Limiting the number of enabled blocks to one active block at a time minimizes the number of bits required for enabling blocks.

Data value in the C•RAM instruction is referenced This micro-instruction alerts the sequencer that the data value is referenced. References can be used for C•RAM instructions that require data values that are wider than the eight bits available in the data field. The reference is stored in the data field and points to a register in the data register file. The referenced value is retrieved and inserted in the C•RAM instruction before it is issued.

Data value and row address in the C•RAM instruction are referenced This micro-instruction alerts the sequencer that both the row address and data fields are referenced in the C•RAM instruction. The references for the row address and data value are stored in the row address and data fields respectively. The referenced values are retrieved from the register files and inserted in the C•RAM instruction before it is issued.

Beginning of a loop This micro-instruction is used to alert the sequencer that the current point in the code store is the beginning of a loop, and loop iterations will begin from this point. To simplify the condition checks for this micro-instruction it is not issued with a C•RAM instruction.

End of a loop Marking the end of a loop this micro-instruction is used to signal the end of the current iteration of the loop, or the end of iterations if the maximum number is reached. Like the micro-instruction marking the beginning of a loop there is no C•RAM instruction issued.

B.3 Sequencer Design

Overview

The sequencer is designed to be able to maintain a high instruction issue rate to the C•RAM device. To support this goal two additional components are used: a C•RAM instruction FIFO buffer, and a C•RAM interface unit, see Figure B.2. The interface unit handles the issuing of instructions to the C•RAM device and generates the required control signals. The instructions that it issues to the C•RAM device are extracted from the FIFO between the sequencer and the interface unit. As it does not have to do any sanity checking of C•RAM instructions, the interface unit can issue C•RAM instructions at a maximum rate of one instruction every two clock cycles (one for generating RAS and the second for generating CAS).

Since the sequencer can take advantage of pipelining, it is capable of producing an instruction every clock cycle. Some instructions, however, do not contain C•RAM components (see Table B.1), and as a result gaps occur in the C•RAM instruction stream. The FIFO addresses this issue by buffering C•RAM instructions from the sequencer, and disguising gaps in the instruction stream. The probability of the FIFO becoming empty is very low, because the C•RAM interface unit does not have as high an issue rate as the sequencer, thus a constant stream of instructions to the C•RAM device can be maintained.

Address and Data Registers

The sequencer has two register files, one for storing row addresses and another for storing wide data values. The address register file consists of two values per register, a current value and an increment value, both of which are ten bits wide. Each register is self-incrementing such that the current value can be updated according to the following formula:

$$\text{new value} = \text{current value} + \text{increment value}$$

When the sequencer encounters a *increment row address register* micro-instruction, it enables the increment feature of the affected registers. The registers are initialized by the sequencer, when it encounters a *set row address register value* micro-instruction, and both the current value and increment portions of the affected address registers are loaded with new values. The ability to initialize and increment addresses allows the sequencer to better support the use of loops in the SIMD function. The address register file can contain up to 256 registers depending on how the incrementation and initialization mechanisms of the sequencer are implemented.

The data register file is 32-bits wide and is used to hold data constants that are too large to fit in the 8-bit data field. Unlike the address registers, the data registers cannot be loaded by the sequencer and have to be configured by the host before sequencing begins. During data referencing the numerical value of the data field is used to index the data register file. This allows up to 256 data registers to be indexed by the data field value.

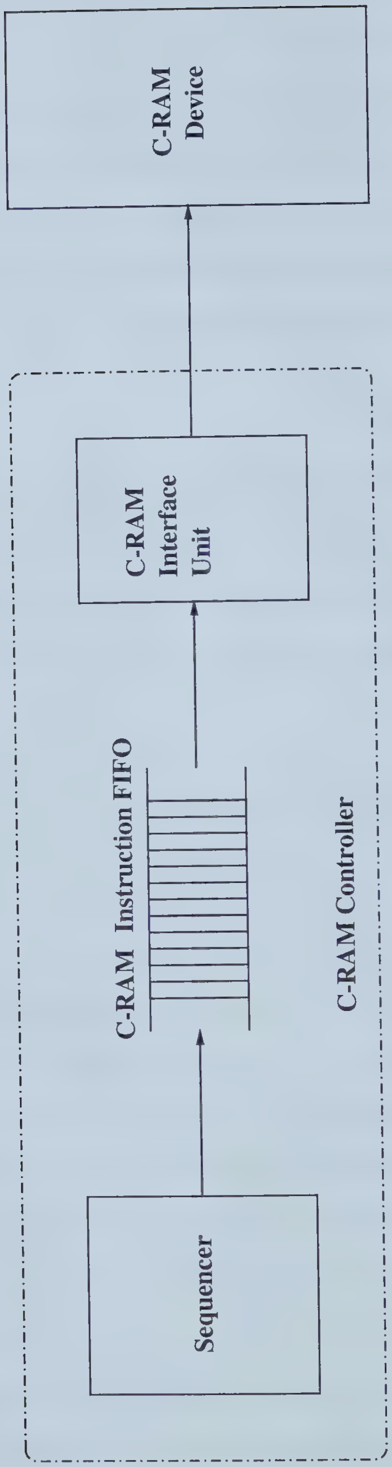

Figure B.2: Controller–based Sequencer

Data Field Value			Register Number
00000001	Row Address	Increment	0
00000010	Row Address	Increment	1
00000100	Row Address	Increment	2
00001000	Row Address	Increment	3
00010000	Row Address	Increment	4
00100000	Row Address	Increment	5
01000000	Row Address	Increment	6
10000000	Row Address	Increment	7

Figure B.3: Sequencer Address Registers with bit-field indexing

Data Field Value		Register
00000000	Data	0
00000001	Data	1
00000010	Data	2
	• • •	
11111110	Data	254
11111111	Data	255

Figure B.4: Sequencer Data Registers

The format of the register files is illustrated in Figures B.3 and B.4.

Operation: Loops

Within a function the host can use multiple levels of loops to support more than one dimension of operation. The number of loop levels supported is based on how the sequencer is implemented, but should be restricted to a maximum level of nesting. Restricting the number of levels supported allows the sequencer to be designed to use a set of registers to track nesting, instead of a stack mechanism. With no stack mechanism the sequencer avoids the overhead of maintaining a stack, and minimizes the impact of loop nesting on the C•RAM instruction stream.

The sequencer uses variable spaces to track which levels of looping are active and how many iterations occur at each level. Prior to beginning the sequencing of the function, the host configures the loop variables and sets the number of iterations for each loop level. Once sequencing has began the micro-instructions used for marking the beginning and end of iterative sections in the function allow the sequencer to track the levels of looping. In Figure B.5 a sequencer with three levels of looping is illustrated.

When the sequencer encounters a micro-instruction marking the beginning of a loop, it stores the address of the first instruction in that section and activates the loop variable for that looping level. As other *beginning of loop* micro-instructions are encountered the sequencer repeats the process, and moves to the next level of looping until all the levels of looping are active. When an *end of loop* micro-instruction is encountered by the sequencer, its checks to see if the iteration limit has been reached for this loop level. If the iteration limit has been reached, the sequencer deactivates the loop variable for this level, and the next instruction after the current one is loaded. If the limit has not been reached, the sequencer loads the first instruction of the loop and increments its iteration measure. Because the sequencer is aware of exactly how many iterations are required at each loop level, only branch hardware for performing iterations is required.

The micro-instructions related to loops, *beginning of loop* and *end of loop*, are not packed with C•RAM instructions. This simplifies the condition checks required when dealing with these micro-instructions but results in no C•RAM instructions being issued during this time. The FIFO will be able to buffer gaps in the C•RAM instruction stream caused by the beginning or ending of a loop section.

Operation: Memory Referencing

The host uses referencing in C•RAM instructions in two situations: where row addresses increase, e.g. in loop iterations; and where data values are too wide for the data field of an instruction, e.g. mask values. The references are stored in their respective fields, the row address field for the row address reference and the data field for the data reference. These are decoded as indices into the appropriate

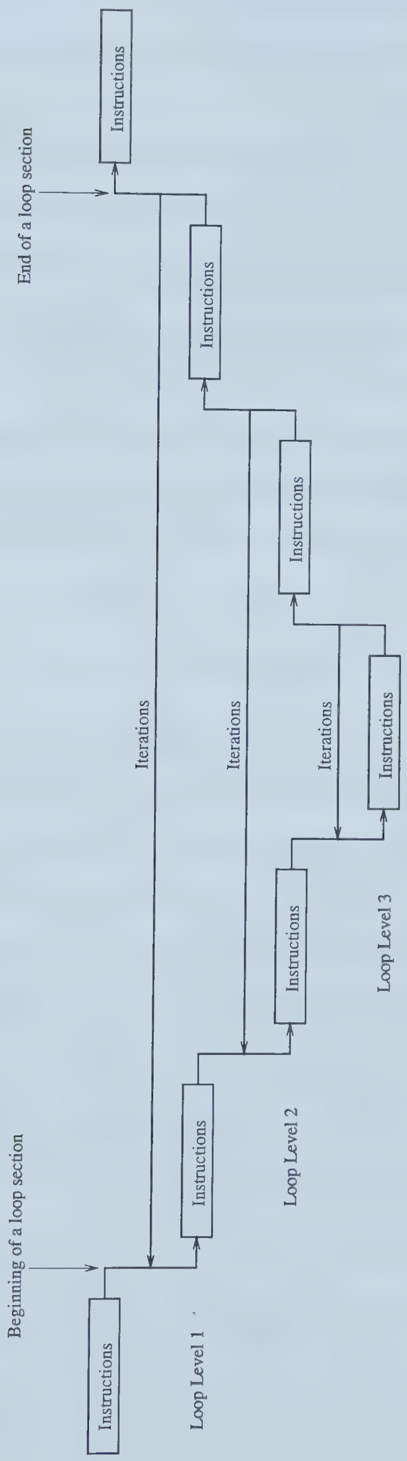

Figure B.5: Sequencer with three loop levels

register file. The sequencer is alerted by *a referenced value* micro-instruction that is packed with the C•RAM instruction. Before the C•RAM instruction is stored in the FIFO, the sequencer accesses the required registers and replaces the references with the referenced values.

The process of retrieving references delays the insertion of the C•RAM instruction in to the FIFO, and with the probability of references occurring being reasonably high the FIFO cannot be expected to buffer delays. Pipelining benefits the retrieval of references as the process can be amortized over multiple stages, and interleaved with the processing of other instructions. Data hazards can occur, however, from the existence of *set register* and *increment register* micro-instructions with *referenced value* micro-instructions in the pipeline. Additional hardware is required to monitor the pipeline and prevent such situations from occurring.

B.4 Summary

The controller–based sequencer is an alternative to the host–based sequencer in the situations where a large latency gap exists between the host and the C•RAM device. While the host is still responsible for generating the SIMD instructions, it can relieve itself of sequencing a long stream of SIMD instructions by using the sequencer on the controller. The penalty of transferring the instructions for a function is reduced by the support of loops in the sequencer, which allows the code to be compacted into repetitive sections. With the controller responsible for sequencing the SIMD instructions the host processor can perform other tasks, while the function is being sequenced to the C•RAM device.

Appendix C

Implementation of the ESM Controller State Machine

C.1 State Machine Signals

The signals used by the state machine are listed in Table C.1.

Table C.1: State Machine Signals

Signal	Direction	Function	Default Value
m32bs_d1	Input	Captured $\overline{BS}$ Signal for one clock period	–
m32bs_dm	Input	Captured $\overline{BS}$ Signal for three clock periods	–
read32	Input	Address decoder signal for a 32-bit read	–
write32	Input	Address decoder signal for a 32-bit write	–
write16	Input	Address decoder signal for a 16-bit write	–
read16	Input	Address decoder signal for a 16-bit read	–
ref_timeout	Input	Time out signal from the Refresh Timer	–
m32burst	Input	M32R/D signal for burst transfers	–
clk	Input	Clock signal	–
reset	Input	Reset signal	–
m32dc	Output	M32R/D signal for declaring the end of a bus cycle	Tri-stated (pulled high by MSA2000 pull-up)

Signal	Direction	Function	Default Value
enable_cramdb	Output	Signal to enable the output buffers to the C•RAM data bus	Low (Disabled)
enable_m32db	Output	Signal to enable the output buffers to the M32R/D data bus	Low (Disabled)
load_msb	Output	Signal to load most significant word of a burst write into the MSB register	Low (Disabled)
cram_ras	Output	RAS signal for the C•RAM device	High (Active low)
cram_cas	Output	CAS signal for the C•RAM device	High (Active low)
cram_we	Output	Write enable for the C•RAM device	High (Active low)
cram_add_select	Output	Signal to switch between row and column addresses in the address path logic	Low (row address selected)
m32db_select	Output	Signal to switch between the MSB and LSB portions of the 32-bit word for the M32R/D data bus	Low (MSB selected)
cram_ref_select	Output	Signal to switch between the refresh row address and the normal address path logic	Low (normal address selected)
inc_ref_add	Output	Signal to increment the refresh address counter	Low (No increment)
led_out (2:0)	Output	Output to LEDs for debugging purposes	–

C.2 State Machine Branches

C.2.1 Reads by M32R/D with A 32-bit Data Item

This operation begins when *m32bs_d1* is asserted, *read32* is driven high, and *m32burst* is asserted.

s1_32r The row address is loaded into the C•RAM device in this state. With RAS low the C•RAM device will latch the row address on the next rising edge of its clock (inverted from the controller clock).

- $cram\_ras = Low$

s2_32r The column address is loaded into the C•RAM device in this state. With CAS low the C•RAM device will latch the column address on the next rising edge of its clock.

- $cram\_ras = Low$
- $cram\_cas = Low$
- $cram\_add\_select = High$ – Selects the column address

s3_32r The data for reading is available by the next rising clock edge of the AX256 after the column address has been latched. The output buffers connected to M32R/D data bus are enabled, so that when the data from AX256 becomes available it passes straight to the M32R/D data bus. The 32-bit word from the C•RAM device has to be split, and the most significant portion put on the M32R/D data bus in this state.

- $cram\_ras = Low$
- $cram\_cas = Low$
- $cram\_add\_select = High$
- $m32dc = Low$ – Signal the end of the first bus cycle
- $enable\_m32db = High$ – Enable output buffers to M32R/D data bus

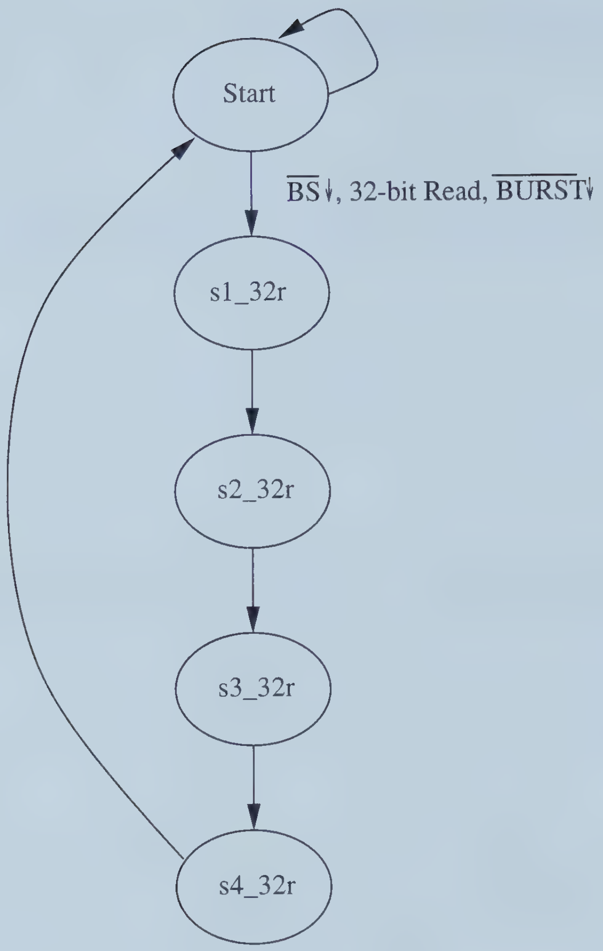
Start
$\overline{BS}\downarrow$, 32-bit Read, $\overline{BURST}\downarrow$
s1_32r
s2_32r
s3_32r
s4_32r

Figure C.1: State Diagram for a 32-bit Read

s4_32r This state is used to put the least significant 16-bit portion of the 32-bit C•RAM word on the M32R/D data bus. As there is no change in the column address, the same 32-bit word remains on C•RAM data bus.

- *cram_ras = Low*
- *cram_cas = Low*
- *cram_add_select = High*
- *m32dc = Low* – Signal the end of the second bus cycle
- *m32db_select = High* – Selects the LSB 16-bit portion of the 32-bit C•RAM word.
- *enable_m32db = High*

C.2.2 Reads by M32R/D with An 8-bit Data Item

This operation begins when *m32bs_d1* is asserted, and *Read16* is driven high.

s1_16r This state is used to load the row address into the C•RAM device. When RAS is driven low the AX256 latches the row address on the next rising edge of its clock.

- *cram_ras = Low*

s2_16r This state is used to load the column address into the C•RAM device. When CAS is driven low the AX256 latches the column address on the next rising edge of its clock.

- *cram_ras = Low*
- *cram_cas = Low*
- *cram_add_select = High* – Select the column address

s3_16r Once the column address has been latched the data will be available by the next rising edge of the AX256 clock. The output buffers connected to the M32R/D data bus are enabled, so that the data passes straight through. As only 16 bits needs to be transferred, only the least significant portion of the 32-bit C•RAM word is passed through.

- *cram_ras = Low*
- *cram_cas = Low*
- *m32dc = Low* – Signal the end of the bus cycle
- *m32db_select = High* – Select the LSB 16-bit portion of the C•RAM word
- *enable_m32db = High* – Enable the output buffers to M32R/D data bus

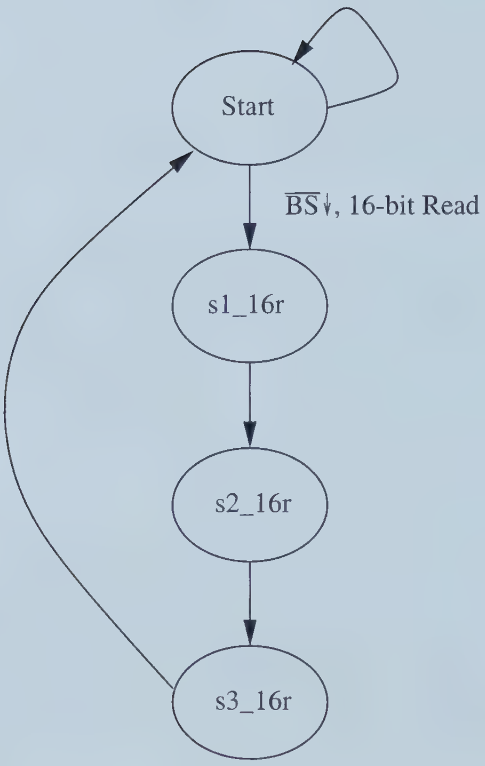

Figure C.2: State Diagram for a 16-bit Read with an 8-bit data

C.2.3 Writes by M32R/D with A 32-bit Data Item

This operation begins when *m32bs_d1* is asserted, *write32* is driven high, and *m32burst* is driven low.

s1_32w Since a 32-bit write is a burst transfer this state loads the MSB portion of the word being transferred, and loads the row address into the C•RAM device. The MSB register, which is enabled in this state, loads the 16-bit word on the falling edge of the controller clock.

- *cram_ras = Low*
- *load_msb = High* – Enable the MSB register
- *m32dc = Low* – Signal the end of the first bus cycle

s2_32w As the most significant portion of the 32-bit word is loaded in the MSB register, this state is used to pass the LSB portion through to be joined with the MSB and form the 32-bit C•RAM word. Both the C•RAM word and the column address are loaded by the AX256 on the rising edge of its clock, as CAS and the write enable (active low) are driven low.

- *cram_ras = Low*
- *cram_cas = Low*
- *cram_add_select = High* – Select column address
- *enable_cramdb = High* – Enable output drivers to the AX256 data bus
- *cram_we = Low* – Signal a write cycle to the AX256 device

C.2.4 Writes by M32R/D with An 8-bit Data Item

This operation begins when *m32b_d1* is asserted, and *write16* is driven high.

s1_16w This state loads row address into the C•RAM device. As only a 16-bit word needs to be transferred the MSB register is not used.

- *cram_ras = Low*

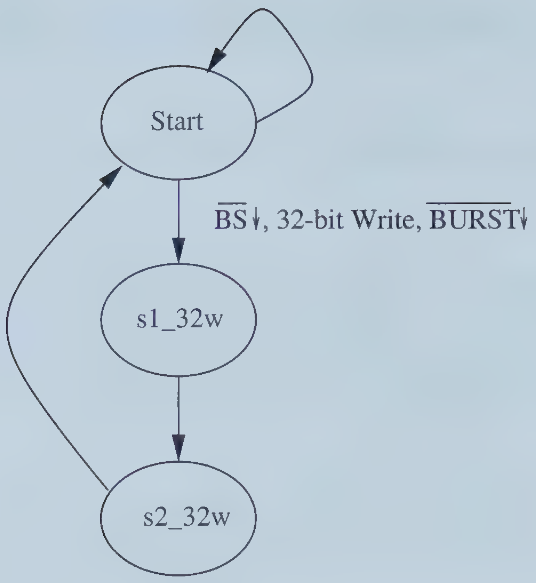
Start
$\overline{BS}\downarrow$, 32-bit Write, $\overline{BURST}\downarrow$
s1_32w
s2_32w

Figure C.3: State Diagram for 32-bit Write

s2_16w The column address for the write is loaded in this state. On the rising edge of its clock the AX256 latches the address and loads the data word.

- $cram\_ras = Low$
- $cram\_cas = Low$
- $cram\_add\_select = High$ – Select column address
- $cram\_we = Low$ – Signal a write cycle to the AX256 device
- $enable\_cramdb = High$ – Enable output buffers to the AX256 data bus
- $m32dc = Low$ – Signal the end of a bus cycle

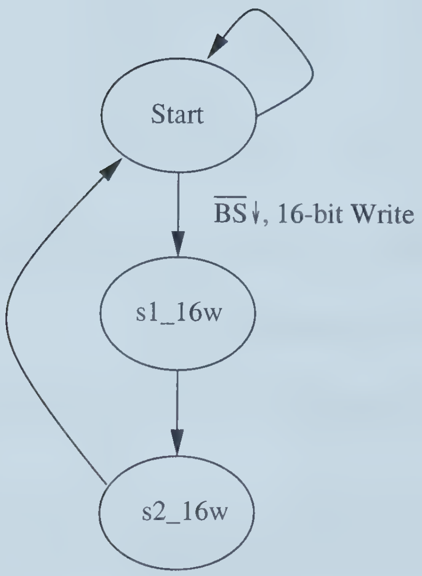

Figure C.4: State Diagram for a 16-bit Write with an 8-bit data

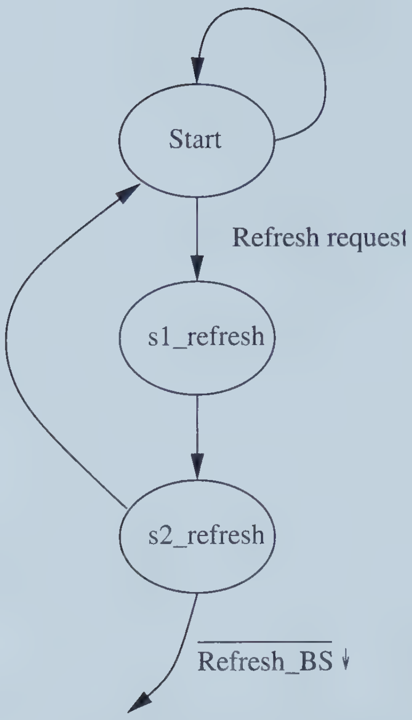

Figure C.5: State Diagram for Refreshes

C.2.5 Refreshing the DRAM Memory Core

This operation is begin when the refresh timer has issued a refresh request by driving *ref_timeout* high.

s1_refresh To refresh a DRAM row all that is required is to enable the affected row by loading the respective row address. This occurs in this state.

- $cram\_ras = Low$
- $cram\_ref\_select = High$

s2_refresh This state serves a similar purpose to the default **start** state and is used to handle the occurrence of a $\overline{\text{BS}}$ assertion by the M32R/D during refresh. Unlike the **start** state this state checks the condition of the *m32bs_dm* signal, which represents a $\overline{\text{BS}}$ assertion held for three clock periods. If *m32bs_dm* is asserted, then a condition check is conducted to see which operation if any is expected, and then branches to the respective state for that operation. In the case that no operation is expected it branches to the **start** state.

Appendix D

C•RAM Daughter-board PCB Plots

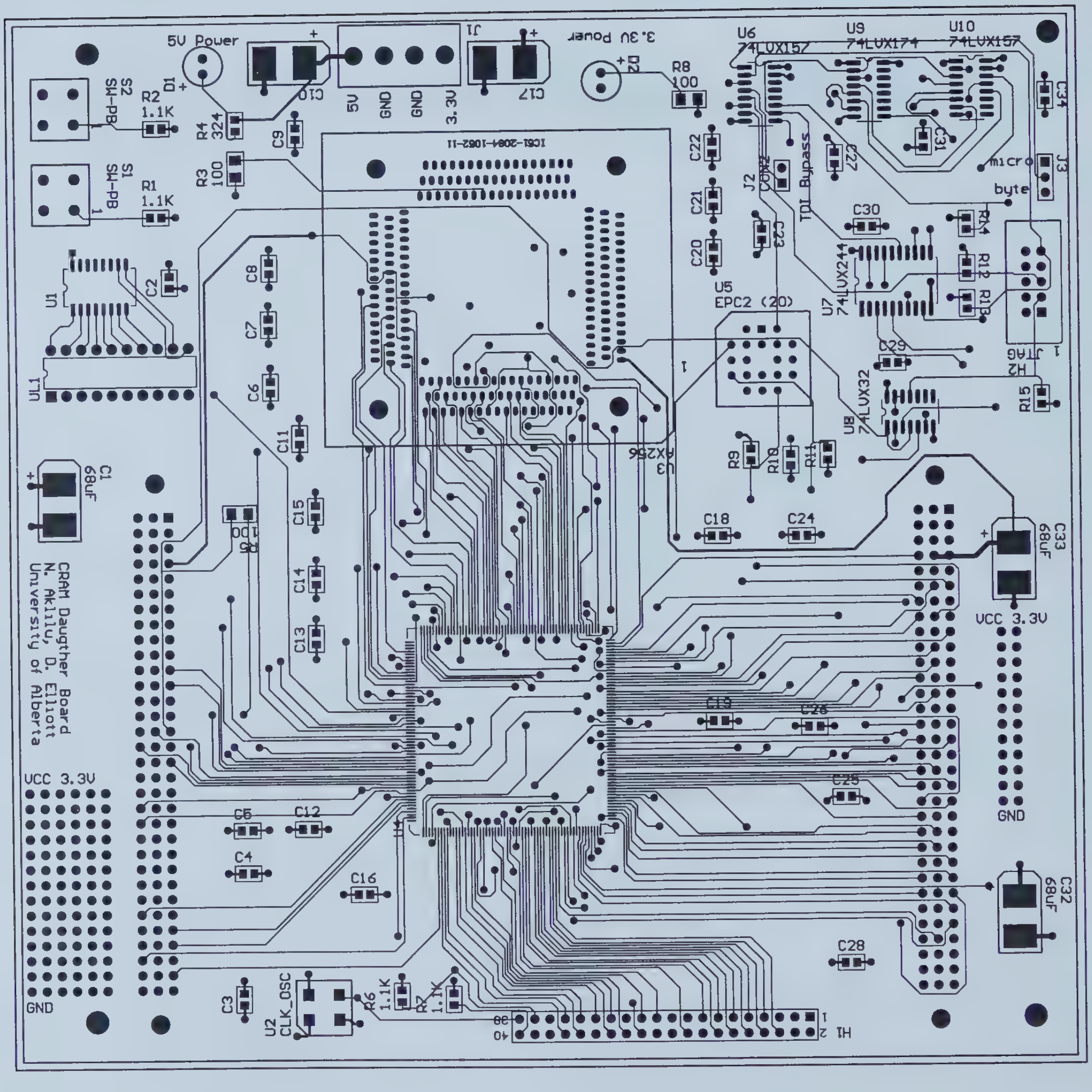

Figure D.1: PCB Top Layer

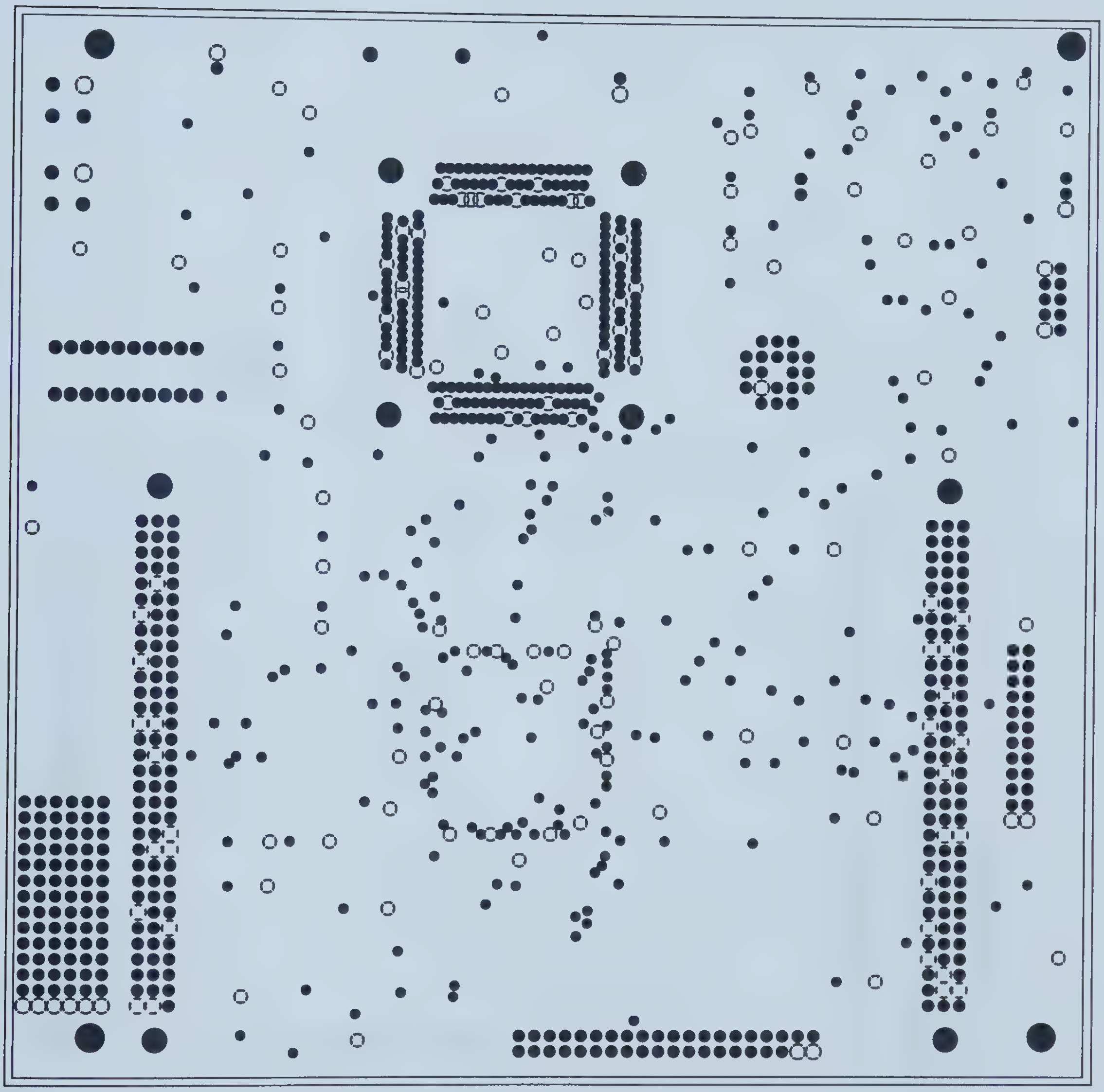

Figure D.2: Internal Ground Layer

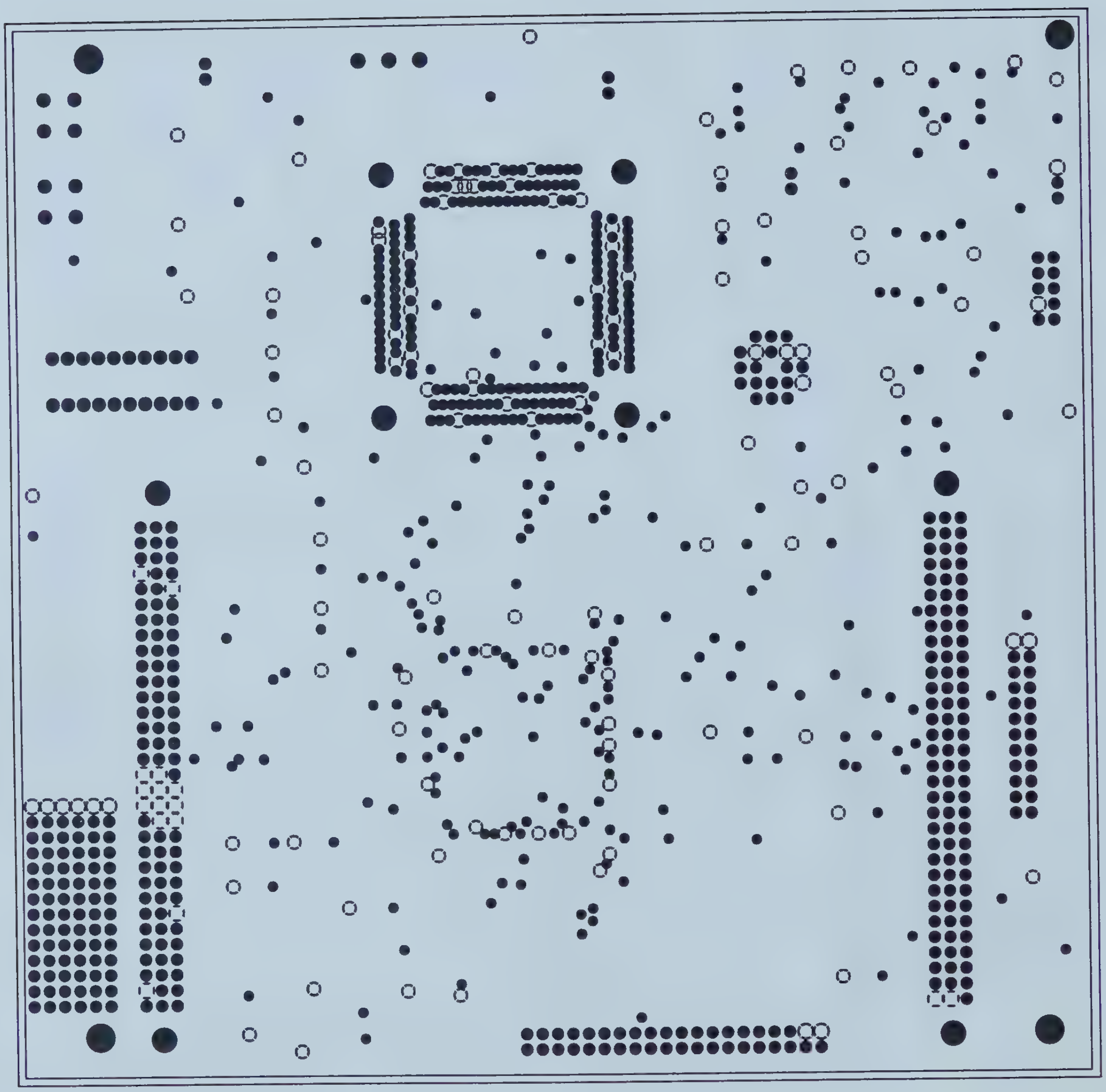
Figure D.3: Internal 3.3V Layer

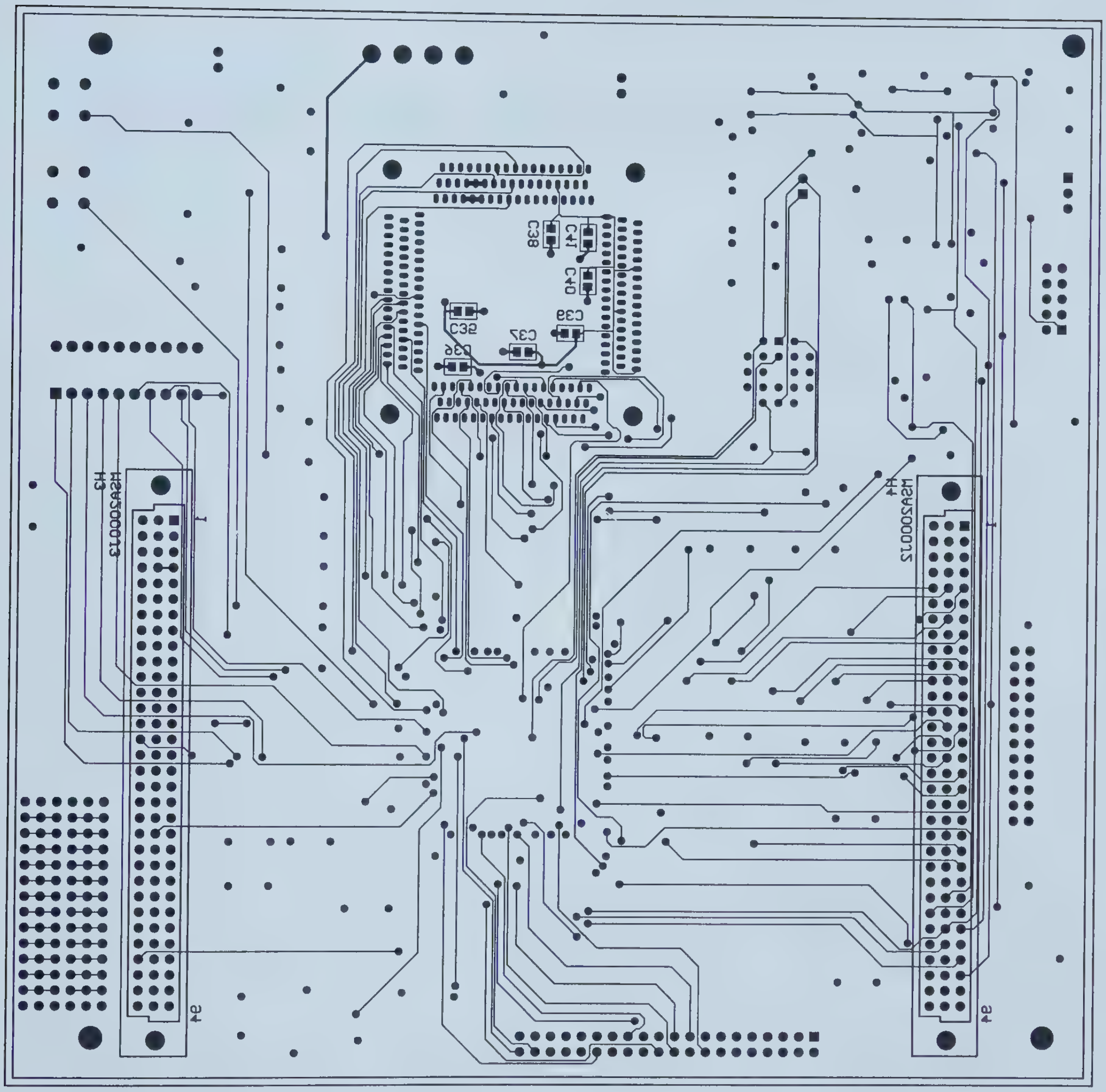

Figure D.4: PCB Bottom Layer

Appendix E

VHDL Code for the ESM C•RAM controller

E.1 Top FPGA Module

```
--****************************************************************
--CRAM Controller
--
-- Copyright (c) 2001 by Noah Aklilu, Edmonton, AB, Canada
-- All rights reserved.
-- This software may be used for non-profit university research
-- if given the author's expressed permission.  An executed license
-- agreement with the author is required for all other uses of
-- this software.  Redistribution of this software is not
-- permitted without the author's expressed permission.
-- This copyright notice must remain intact.
-- Derivative works may contain additional notices.
--
-- This software comes with no warranty.
--
--* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
-- fpga.vhd
--
-- Framework file to encapsulate design and tri-state unused
-- I/O pins
--
--****************************************************************
--

library ieee;
use ieee.std_logic_1164.all;
library altera;
use altera.maxplus2.all;
library work;
use work.comp_pack.all;

entity fpga is
port (

-- Microprocessor Signals --

-- Address Bus
  m32a : in std_logic_vector(8 to 30);

-- Control Signals
  m32sid : in std_logic;
  m32bch : in std_logic;
  m32bcl : in std_logic;
  m32bs  : in std_logic;
  m32st  : in std_logic;
  m32rw : in std_logic;
  m32burst : in std_logic;
  m32dc : out std_logic;
  m32hreq : out std_logic;
  m32hack : in std_logic;
  m32cs : out std_logic;
  m32wkup : out std_logic;
  m32stby : in std_logic;

-- Data Bus
  m32d : inout std_logic_vector(0 to 15);
-- Other
  m32pp0 : inout std_logic;
  m32pp1 : inout std_logic;
  m32rstin : in std_logic;
  m32clkin : in std_logic;

-- M65439BFP ASIC Signals --
  asic_bcs : in std_logic_vector(0 to 3);  -- Chip Select Signals
  asic_bint : out std_logic_vector(0 to 3); -- Interrupt Signals
  asic_bdreq : out std_logic;
  asic_bdack : in std_logic;
  asic_bcswait : out std_logic;
  asic_bcswrl : in std_logic;
  asic_bcswrh : in std_logic;
  asic_bcsrd : in std_logic;
  asic_ultxd : in std_logic;
  asic_ulrxd : out std_logic;
  asic_busbufen : in std_logic;
  asic_bport00 : inout std_logic;

--CRAM Signals --

-- Address Bus
  cram_ad : out std_logic_vector(9 downto 0);

-- Data Bus
  cram_db : inout std_logic_vector(31 downto 0);

-- Control
  cram_test : out std_logic_vector(2 downto 0);
  cram_tri_en : out std_logic;
  cram_test_out : in std_logic;
  cram_ras : out std_logic;
  cram_cas : out std_logic;
  cram_we : out std_logic;
  cram_wm : out std_logic_vector(3 downto 0);
  cram_reset : out std_logic;

-- Clocks
  cram_sclk : out std_logic;
  cram_dpclk : out std_logic;

-- Onboard Stuff ---

-- Clock Oscillator
  fpga_gclk2 : in std_logic;

-- Header
  fpga_45 : inout std_logic;
  fpga_44 : inout std_logic;
  fpga_43 : inout std_logic;
  fpga_41 : inout std_logic;
  fpga_40 : inout std_logic;
  fpga_39 : inout std_logic;
  fpga_38 : inout std_logic;
  fpga_36 : inout std_logic;
  fpga_35 : inout std_logic;
  fpga_34 : inout std_logic;
  fpga_33 : inout std_logic;
  fpga_31 : inout std_logic;
  fpga_30 : inout std_logic;
  fpga_29 : inout std_logic;
  fpga_28 : inout std_logic;
  fpga_25 : inout std_logic;
  fpga_24 : inout std_logic;
  fpga_21 : inout std_logic;
  fpga_20 : inout std_logic;
  fpga_19 : inout std_logic;
  fpga_18 : inout std_logic;
  fpga_17 : inout std_logic;
  fpga_15 : inout std_logic;
  fpga_14 : inout std_logic;
  fpga_13 : inout std_logic;
  fpga_12 : inout std_logic;
  fpga_9 : inout std_logic;
  fpga_8 : inout std_logic;
  fpga_7 : inout std_logic;
  fpga_in : in std_logic;

-- LEDs
  fpga_led : out std_logic_vector(9 downto 0);

-- Push Buttons
  fpga_pb1 : in std_logic;
  fpga_pb2 : in std_logic
);
end entity;

architecture toplevel of fpga is

signal int_dc : std_logic;
signal int_enable_dc : std_logic;

begin

--Comment out what you use

--Control Signals
--  m32dc <= 'Z';
  m32hreq <= 'Z';
  m32cs <='Z';
  m32wkup <= 'Z';

--Data Bus
--  m32d <= (others => 'Z');

--Other
  m32pp0 <= 'Z';
  m32pp1 <= 'Z';

--M65439BFP ASIC Signals--
  asic_bint <= (others => 'Z'); -- Interrupt Signals
  asic_bdreq <= 'Z';
  asic_bcswait <= 'Z';
  asic_u1rxd <= 'Z';
  asic_bport00 <= 'Z';

--CRAM Signals--

--Address Bus
--  cram_ad <= (others => 'Z');

--Data Bus
--  cram_db <= (others => 'Z');

--Control
--  cram_test <= (others => 'Z');
--  cram_tri_en <= 'Z';
--  cram_ras <= 'Z';
--  cram_cas <= 'Z';
--  cram_we <= 'Z';
--  cram_wm <= (others => 'Z');
--  cram_reset <= 'Z';

--Clocks
  cram_sclk <= '0';
--  cram_sclk <= 'Z';
--  cram_dpclk <= 'Z';

--Onboard Stuff---

--Header
  fpga_45 <= 'Z';
  fpga_44 <= 'Z';
  fpga_43 <= 'Z';
  fpga_41 <= 'Z';
  fpga_40 <= 'Z';
  fpga_39 <= 'Z';
  fpga_38 <= 'Z';
  fpga_36 <= 'Z';
  fpga_35 <= 'Z';
  fpga_34 <= 'Z';
  fpga_33 <= 'Z';
  fpga_31 <= 'Z';
  fpga_30 <= 'Z';
  fpga_29 <= 'Z';
  fpga_28 <= 'Z';
  fpga_25 <= 'Z';
  fpga_24 <= 'Z';
  fpga_21 <= 'Z';
  fpga_20 <= 'Z';
  fpga_19 <= 'Z';
  fpga_18 <= 'Z';
  fpga_17 <= 'Z';
  fpga_15 <= 'Z';
  fpga_14 <= 'Z';
  fpga_13 <= 'Z';
  fpga_12 <= 'Z';
  fpga_9 <= 'Z';
  fpga_8 <= 'Z';
  fpga_7 <= 'Z';

--LEDs
--  fpga_led <= (others => 'Z');

----------------------------------------------------------------
--The data complete signal "dc" needs to be driven high after
--being driven low (assertion). This logic is for that purpose
--

        dc_tristate : componenet TRI
          port map (
            a_in  => int_dc,
            oe      => int_enable_dc ,
            a_out => m32dc );

        dc_enable_reg  :  process(m32clkin) is
        begin
          if(falling_edge(m32clkin)) then
            int_enable_dc <= not int_dc;
          end if;
        end process;

-------------------------------------------------------------------------------


-- Simple controller

  simple_controller  :  component controller
    port map (

    -- M32R/D Interface
      m32sid => m32sid,
      m32burst => m32burst,
      m32rw => m32rw,
      m32bch => m32bch,
      m32bcl => m32bcl,
      m32add => m32a,
      m32db_in => m32d,
      m32db_out => m32d,
      m32dc => int_dc,
      m32clkin => m32clkin,
      m32bs => m32bs,

    -- CRAM Interface

      cram_cas => cram_cas,
      cram_ras => cram_ras,
      cram_add => cram_ad,
      cram_db_in => cram_db,
      cram_db_out => cram_db,
      cram_we => cram_we,
      cram_wmmask => cram_wm,
      cram_test => cram_test,
      cram_reset => cram_reset,
      cram_tri_en => cram_tri_en,
--      cram_sclk => cram_sclk,
      cram_dpclk => cram_dpclk,

    -- Controller reset
      reset => fpga_pb1,

    -- Debugging Output onto leds

      led_out => fpga_led
    );

end architecture;
```

E.2 Controller Module

```
--*****************************************************************
--CRAM Controller
--
-- Copyright (c) 2001 by Noah Aklilu, Edmonton, AB, Canada
-- All rights reserved.
-- This software may be used for non-profit university research
-- if given the author's expressed permission.  An executed license
-- agreement with the author is required for all other uses of
-- this software.  Redistribution of this software is not
-- permitted without the author's expressed permission.
-- This copyright notice must remain intact.
-- Derivative works may contain additional notices.
--
-- This software comes with no warranty.
--
--* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
-- controller.vhd
--
-- The encapsulating entity for the controller components
--
--******************************************************************
--

library ieee;
use ieee.std_logic_1164.all;
library work;
use work.comp_pack.all;

entity controller is
port (

--M32R/D Interface
```

```
  m32sid : in std_logic;
  m32burst : in std_logic;
  m32rw : in std_logic;
  m32bch : in std_logic;
  m32bcl : in std_logic;
  m32add : in std_logic_vector(8 to 30);
  m32db_in : in std_logic_vector(0 to 15);
  m32db_out : out std_logic_vector(0 to 15);
  m32dc : out std_logic;
  m32clkin : in std_logic;
  m32bs : in std_logic;

--CRAM Interface

  cram_cas : out std_logic;
  cram_ras : out std_logic;
  cram_add : out std_logic_vector(9 downto 0);
  cram_db_in : in std_logic_vector(31 downto 0);
  cram_db_out : out std_logic_vector(31 downto 0);
  cram_we : out std_logic;
  cram_wmmask : out std_logic_vector(3 downto 0);
  cram_test : out std_logic_vector(2 downto 0);
  cram_reset : out std_logic;
  cram_tri_en : out std_logic;
  cram_sclk : out std_logic;
  cram_dpclk : out std_logic;

-- Controller reset
  reset : in std_logic;

-- Debugging Output onto leds

  led_out : out std_logic_vector(9 downto 0)
);
end entity;

architecture structural of controller is

signal int_read32,int_write32,int_read16,int_write16 : std_logic;
signal int_cram_add_select : std_logic;
signal inv_m32clkin : std_logic;
signal int_bs_d1,int_bs_dm : std_logic;
signal int_enable_m32db,int_load_msb,int_enable_cramdb,int_data_word_sel : std_logic;
signal int_cram_add : std_logic_vector(9 downto 0);
signal int_ref_add : std_logic_vector(9 downto 0);
signal int_ref_add_sel : std_logic;
signal int_inc_ref_add : std_logic;
signal int_ref_timeout : std_logic;
signal int_led_out : std_logic_vector(3 downto 0);
signal short_m32add : std_logic_vector(13 to 30);
signal int_ppu_ctrl_reg : std_logic;
signal int_ppu_dram_transfer : std_logic;

begin

-- Smaller Address Bus used by translation block
  short_m32add <= m32add(13 to 30);

-- Inverted Clock for most components
  inv_m32clkin <= not m32clkin;

--CRAM Standard Constant Stuff
  cram_test <= ”000”;
  cram_tri_en <= ’0’;
  cram_reset <= reset;

-- Should only be for Chunky mode but you never know
  cram_wmmask <= ”1111”;

--CRAM clk
  cram_sclk <= inv_m32clkin;
  cram_dpclk <= inv_m32clkin;

--LED constants
  led_out(9) <= ’1’;
  led_out(8 downto 4) <= ”00000”;
  led_out(3 downto 0) <= int_led_out;

--Mux for switching between Refresh Address and Normal Addressing

  with int_ref_add_sel select
    cram_add <= int_ref_add when ’1’,
          int_cram_add when others;

-- Address Decoder for State Machine

  add_decod1 : component add_decode
  port map (
    add_bus => m32add,
    rw => m32rw,
    sid => m32sid,
    bch => m32bch,
      bcl => m32bcl,
      ppu_dram_transfer => int_ppu_dram_transfer,
      ppu_ctrl_reg => int_ppu_ctrl_reg,
      read32 => int_read32,
      write32 => int_write32,
      read16 => int_read16,
      write16 => int_write16
    );


  -- Logic for the Address Path

    addbus_path1 : component addbus_path
    port map (
      m32add => short_m32add,
      cramadd => int_cram_add,
      add_select => int_cram_add_select
    );


  -- BS Delay block

    bs_delay : component bs_reg
    port map (
      m32bs => m32bs,
      bs_dl => int_bs_dl,
      bs_dm => int_bs_dm,
      clk => m32clkin,
      reset => reset
    );


  -- Logic for Data Bus flow

    databus_path1 : component databus_path
      port map (
        m32db_in => m32db_in,
        m32db_out => m32db_out,
        enable_m32db => int_enable_m32db,
        load_msb => int_load_msb,
        ppu_dram_transfer => int_ppu_dram_transfer,
        ppu_ctrl_reg => int_ppu_ctrl_reg,
        clk => m32clkin,
        reset => reset,
        cramdb_in => cram_db_in,
        cramdb_out => cram_db_out,
        enable_cramdb => int_enable_cramdb,
        data_word_sel => int_data_word_sel
      );


  -- Refresh Monitor

    refresh_mon1 : component refresh_mon
    port map (
      clk => m32clkin,
      reset => reset,
      inc_ref_add => int_inc_ref_add,
      ref_add => int_ref_add,
      ref_timeout => int_ref_timeout
    );


  -- Controller State Machine

    statemachine1 : component statemachine
    port map (
      m32bs_dl => int_bs_dl,
      m32bs_dm => int_bs_dm,
      read32 => int_read32,
      write32 => int_write32,
      read16 => int_read16,
      write16 => int_write16,
      ref_timeout => int_ref_timeout,
      m32burst => m32burst,
      clk => m32clkin,
      reset => reset,
      m32dc => m32dc,
      enable_cramdb => int_enable_cramdb,
      enable_m32db => int_enable_m32db,
      load_msb => int_load_msb,
      cram_ras => cram_ras,
      cram_cas => cram_cas,
      cram_we => cram_we,
      cram_add_select => int_cram_add_select,
      m32db_select => int_data_word_sel,
      cram_ref_select => int_ref_add_sel,
      inc_ref_add => int_inc_ref_add,
      led_out => int_led_out
    );

  end architecture;
```

E.3 Component Package

```
--***************************************************************
--CRAM Controller
--
-- Copyright (c) 2001 by Noah Aklilu, Edmonton, AB, Canada
-- All rights reserved.
-- This software may be used for non-profit university research
-- if given the author's expressed permission. An executed license
-- agreement with the author is required for all other uses of
-- this software. Redistribution of this software is not
-- permitted without the author's expressed permission.
-- This copyright notice must remain intact.
-- Derivative works may contain additional notices.
--
-- This software comes with no warranty.
--
--* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
-- comp_pack.vhd
--
-- The package for the components of the controller.
--
--*****************************************************************
--

library ieee;
use ieee.std_logic_1164.all;

package comp_pack is

  component add_decode is
    port (

    --M32R signals
      add_bus : in std_logic_vector(8 to 30);
      rw : in std_logic;
      sid : in std_logic;
      bch,bcl : in std_logic;

    -- Data Path Logic Masking info
      ppu_ctrl_reg : out std_logic;
      ppu_dram_transfer : out std_logic;

    -- State Machine
      read32 : out std_logic;
      write32 : out std_logic;
      read16 : out std_logic;
      write16 : out std_logic
    );
  end component;

  component translation_block is
    port (
    -- Microprocessor address bus
      add_bus : in std_logic_vector(13 to 30);

    --Row and Column address outputs
      row_add : out std_logic_vector(9 downto 0);
      col_add : out std_logic_vector(9 downto 0)
    );
  end component;

  component addbus_path is
    port (
      m32add : in std_logic_vector(13 to 30);
      cramadd : out std_logic_vector(9 downto 0);

    -- Used to select whether col or row address is pumped out

      add_select : in std_logic

    );
  end component;

  component bs_reg is
    port(
      m32bs : in std_logic;

    --BS delayed by one clock
      bs_d1 : out std_logic;

    --BS delayed by one or more clocks for refresh condition

      bs_dm : out std_logic;

      clk : in std_logic;
      reset : in std_logic
    );
  end component;

  component controller is
```

```
    port (

    --M32R/D Interface
      m32sid : in std_logic;
      m32burst : in std_logic;
      m32rw : in std_logic;
      m32bch : in std_logic;
      m32bcl : in std_logic;
      m32add : in std_logic_vector(8 to 30);
      m32db_in : in std_logic_vector(0 to 15);
      m32db_out : out std_logic_vector(0 to 15);
      m32dc : out std_logic;
      m32clkin : in std_logic;
      m32bs : in std_logic;

    --CRAM Interface

      cram_cas : out std_logic;
      cram_ras : out std_logic;
      cram_add : out std_logic_vector(9 downto 0);
      cram_db_in : in std_logic_vector(31 downto 0);
      cram_db_out : out std_logic_vector(31 downto 0);
      cram_we : out std_logic;
      cram_wmmask : out std_logic_vector(3 downto 0);
      cram_test : out std_logic_vector(2 downto 0);
      cram_reset : out std_logic;
      cram_tri_en : out std_logic;
      cram_sclk : out std_logic;
      cram_dpclk : out std_logic;

    -- Controller reset
      reset : in std_logic;

    -- Debugging Output onto leds

      led_out : out std_logic_vector(9 downto 0)
    );
  end component;

  component databus_path is
    port (
      m32db_in : in std_logic_vector(0 to 15);
      m32db_out : out std_logic_vector(0 to 15);
      enable_m32db : in std_logic;
      load_msb : in std_logic;
      ppu_dram_transfer : in std_logic;
      ppu_ctrl_reg : in std_logic;
      clk : in std_logic;
      reset : in std_logic;
      cramdb_in : in std_logic_vector(31 downto 0);
      cramdb_out : out std_logic_vector(31 downto 0);
      enable_cramdb : in std_logic;
      data_word_sel : in std_logic
    );
  end component;

  component refresh_mon is
    port (
      clk : in std_logic;
      reset : in std_logic;

    -- Increment Refresh Address signal
      inc_ref_add : in std_logic;

    -- Current Refresh Address
      ref_add : out std_logic_vector(9 downto 0);

    -- Refresh Timeout signal
      ref_timeout : out std_logic
    );
  end component;

  component statemachine is
    port (
    -- Delayed BS signals
      m32bs_d1 : in std_logic;
      m32bs_dm : in std_logic;

    -- Address Decoder Signals
      read32 : in std_logic;
      write32 : in std_logic;
      read16 : in std_logic;
      write16 : in std_logic;

    -- Refresh Timeout
      ref_timeout : in std_logic;

    --M32R burst signal for 32 bit reads and writes
      m32burst : in std_logic;

    -- Self-evident
      clk,reset : in std_logic;
    --M32R Data Complete signal
      m32dc : out std_logic;

    -- Enable CRAM DB output buffer
      enable_cramdb : out std_logic;

    -- Enable M32R DB output buffer
      enable_m32db : out std_logic;

    -- Signal to load MSB during 32 bit writes
      load_msb : out std_logic;

    --CRAM RAS and CAS signals
      cram_ras, cram_cas : out std_logic;

    --CRAM Write Enable Signal
      cram_we : out std_logic;

    -- Switch between Row and Col addresses
      cram_add_select : out std_logic;

    -- Switch between MSB and LSB portions of CRAM word for M32R bus
      m32db_select : out std_logic;

    -- Switch Refresh Address and Normal addressing
      cram_ref_select : out std_logic;

    -- Increment Refresh Address counter
      inc_ref_add : out std_logic;

    -- Debugging Signals
      led_out : out std_logic_vector(3 downto 0)

    );
  end component;

end package;
```

E.4 Address Decoder Module

```
--*****************************************************************
--CRAM Controller
--
-- Copyright (c) 2001 by Noah Aklilu, Edmonton, AB, Canada
-- All rights reserved.
-- This software may be used for non-profit university research
-- if given the author's expressed permission.  An executed license
-- agreement with the author is required for all other uses of
-- this software.  Redistribution of this software is not
-- permitted without the author's expressed permission.
-- This copyright notice must remain intact.
-- Derivative works may contain additional notices.
--
-- This software comes with no warranty.
--
--* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
-- add_decode.vhd
--
-- Contains the code for the address decoder that signals the state
-- machine
--
--*****************************************************************
--

library ieee;
use ieee.std_logic_1164.all;
library work;
use work.comp_pack.all;

entity add_decode is
port (

--M32R signals
  add_bus : in std_logic_vector(8 to 30);
  rw : in std_logic;
  sid : in std_logic;
  bch, bcl : in std_logic; -- Signals that mark upper and lower byte valid

-- Data Path Logic Masking info
  ppu_ctrl_reg : out std_logic;
  ppu_dram_transfer : out std_logic;

-- State Machine
  read32 : out std_logic;
  write32 : out std_logic;
  read16 : out std_logic;
  write16 : out std_logic
);
end entity;
```

```
architecture beh of add_decode is

-- Base address
constant myaddress : std_logic_vector(8 to 12) := "01100";

signal int_trigger : std_logic_vector(8 to 32);
signal int_byte_control : std_logic;

begin

  int_trigger(8 to 30) <= add_bus(8 to 30);
  int_trigger(31) <= sid;
  int_trigger(32) <= rw;
  int_byte_control <= bch or bcl;

--
-- Deals with State Machine signals
--

  main : process(int_trigger)
  begin

-- Make sure it is in the I/O memory space

    if (sid = '1') and (add_bus(8 to 12) = myaddress) and (int_byte_control = '0') then

-- DRAM Memory Access

      if ((add_bus(13 to 22) >= "0000000000" ) and (add_bus(13 to 22) <= "1100101111")) and (add_bus
           (30) = '0') then

-- Reading from DRAM and not a PPU<->DRAM transfer

      if rw = '1' then

          write32 <= '0';
          read32 <= '1';

-- Writing to DRAM

        elsif rw = '0' then

          write32 <= '1';
          read32 <= '0';

        end if;

        write16 <= '0';
        read16 <= '0';

-- SRC PPU Reg

      elsif (add_bus(13 to 22) = "1100110000" ) and (add_bus(30) = '0') then

-- Writing to SRAM

        if rw = '1' then

          write32 <= '0';
          read32 <= '1';

-- Reading from SRAM

        else

          write32 <= '1';
          read32 <= '0';

        end if;

        write16 <= '0';
        read16 <= '0';

-- DEST PPU Reg

      elsif (add_bus(13 to 22) = "1100110001" ) and (add_bus(30) = '0') then

-- Writing to SRAM

        if rw = '1' then

          write32 <= '0';
          read32 <= '1';

-- Reading from SRAM

        else

          write32 <= '1';
          read32 <= '0';
```

```
            end if;

            write16 <= '0';
            read16 <= '0';

    --BRUSH PPU Reg

          elsif (add_bus(13 to 22) = "1100110010" ) and (add_bus(30) = '0') then

    -- Writing to SRAM

            if rw = '1' then

              write32 <= '0';
              read32 <= '1';

    -- Reading from SRAM

            else

              write32 <= '1';
              read32 <= '0';

            end if;

            write16 <= '0';
            read16 <= '0';

    --MASK PPU Reg

          elsif (add_bus(13 to 22) = "1100110011" ) and (add_bus(30) = '0') then

    -- Writing to SRAM

            if rw = '1' then

              write32 <= '0';
              read32 <= '1';

    -- Reading from SRAM

            else

              write32 <= '1';
              read32 <= '0';

            end if;

            write16 <= '0';
            read16 <= '0';

    --ROP PPU Reg

          elsif (add_bus(13 to 22) = "1100110100" ) and (add_bus(30) = '0') then

    -- Writing to SRAM

            if rw = '1' then

              write32 <= '0';
              read32 <= '1';

    -- Reading from SRAM

            else

              write32 <= '1';
              read32 <= '0';

            end if;

            write16 <= '0';
            read16 <= '0';

    --FS <-> PPU Reg

          elsif ((add_bus(13 to 29) >= "11001101010000000") and (add_bus(13 to 29) <= "11001101010001111")) and (
              rw = '0') and (add_bus(30) = '0') then

            write16 <= '1';
            read16 <= '0';
            write32 <= '0';
            read32 <= '0';

    --PPU CTRL Reg

      -- Writing Value
          elsif (add_bus(13 to 30) = "110011010100100000") and (rw = '0') then

            write16 <= '1';
            read16 <= '0';
          write32 <= '0';
          read32 <= '0';

    -- Reading Value
        elsif (add_bus(13 to 30) = "110011010100100000") and (rw = '1') then

          write16 <= '0';
          read16 <= '1';
          write32 <= '0';
          read32 <= '0';


--BANKEN Reg

    -- Writing Value
        elsif (add_bus(13 to 30) = "110011010100100001") and (rw = '0') then

          write16 <= '1';
          read16 <= '0';
          write32 <= '0';
          read32 <= '0';

    -- Reading Value
        elsif (add_bus(13 to 30) = "110011010100100001") and (rw = '1') then

          write16 <= '0';
          read16 <= '1';
          write32 <= '0';
          read32 <= '0';

--RASTOP Reg

    -- Writing Value
        elsif (add_bus(13 to 30) = "110011010100100010") and (rw = '0') then

          write16 <= '1';
          read16 <= '0';
          write32 <= '0';
          read32 <= '0';

    -- Reading Value
        elsif (add_bus(13 to 30) = "110011010100100010") and (rw = '1') then

          write16 <= '0';
          read16 <= '1';
          write32 <= '0';
          read32 <= '0';

--ROPFS

        elsif (add_bus(13 to 30) = "110011010100100011") and (rw = '0') then

          write16 <= '1';
          read16 <= '0';
          write32 <= '0';
          read32 <= '0';

--MASKSYS

        elsif (add_bus(13 to 30) = "110011010100100100") and (rw = '0') then

          write16 <= '0';
          read16 <= '0';
          write32 <= '1';
          read32 <= '0';

--PPU Transfer Reg

        elsif (add_bus(13 to 30) = "110011010100100110") and (rw = '0') then

          write16 <= '1';
          read16 <= '0';
          write32 <= '0';
          read32 <= '0';


-- Funnel Shift Ctrl Reg

    -- Writing Value
        elsif (add_bus(13 to 30) = "110011010100100111") and (rw = '0') then

          write16 <= '1';
          read16 <= '0';
          write32 <= '0';
          read32 <= '0';

    -- Reading Value
        elsif (add_bus(13 to 30) = "110011010100100111") and (rw = '1') then

          write16 <= '0';
          read16 <= '1';
          write32 <= '0';
        read32 <= '0';


-- Funnel Shift Access Reg

-- Write Access

      elsif (add_bus(13 to 29) >= "11001101010010101") and (add_bus(13 to 29) <= "11001101010010111") and (
            add_bus(30) = '0') and (rw = '0') then

        write16 <= '0';
        read16 <= '0';
        write32 <= '1';
        read32 <= '0';

-- Read Access

      elsif ((add_bus(13 to 29) = "11001101010010101") or (add_bus(13 to 29) = "11001101010010110")) and (
            add_bus(30) = '0') and (rw = '1') then

        write16 <= '0';
        read16 <= '0';
        write32 <= '0';
        read32 <= '1';


-- Nothing important happened

      else

        write16 <= '0';
        read16 <= '0';
        write32 <= '0';
        read32 <= '0';

      end if;


    else

      read32 <= '0';
      write32 <= '0';
      write16 <= '0';
      read16 <= '0';

    end if;

  end process;

--
-- Information for the data path logic
--

  data_main : process(int_trigger) is
  begin

-- PPU Control Register

    if (add_bus(13 to 30) = "110011010100100000") and (rw = '0') then

      ppu_ctrl_reg <= '1';

    else

      ppu_ctrl_reg <= '0';

    end if;

-- PPU<->DRAM Transfers
    if ((add_bus(13 to 22) >= "0000000000" ) and (add_bus(13 to 22) <= "1100101111")) and (add_bus(23) = '1')
          and (rw = '0') then

      ppu_dram_transfer <= '1';

    else

      ppu_dram_transfer <= '0';

    end if;

  end process;

end architecture;
```

E.5 Address Path Module

```
--*****************************************************************
-- CRAM Controller
--
-- Copyright (c) 2001 by Noah Aklilu, Edmonton, AB, Canada
-- All rights reserved.
-- This software may be used for non-profit university research
-- if given the author's expressed permission. An executed license
-- agreement with the author is required for all other uses of
-- this software. Redistribution of this software is not
-- permitted without the author's expressed permission.
-- This copyright notice must remain intact.
-- Derivative works may contain additional notices.
--
-- This software comes with no warranty.
--
--* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
-- addbus_path.vhd
--
-- Contains the path from the M32R address bus to the CRAM
-- address bus
--
--*****************************************************************
--

library ieee;
use ieee.std_logic_1164.all;
library work;
use work.comp_pack.all;

entity addbus_path is
port (
  m32add : in std_logic_vector(13 to 30);
  cramadd : out std_logic_vector(9 downto 0);

-- Used to select whether col or row address is pumped out

  add_select : in std_logic

);
end entity;

architecture structural of addbus_path is

signal int_col,int_row : std_logic_vector(9 downto 0);

begin

-- Used to switch between row and col addresses

  with add_select select
    cramadd <= int_col when '1',
           int_row when others;


-- Translation block;

  translator : component translation_block
  port map (
    add_bus => m32add,
    row_add => int_row,
    col_add => int_col
  );

end architecture;
```

E.6 M32R/D $\overline{\text{BS}}$ Handler Module

```
--*****************************************************************
--CRAM Controller
--
-- Copyright (c) 2001 by Noah Aklilu, Edmonton, AB, Canada
-- All rights reserved.
-- This software may be used for non-profit university research
-- if given the author's expressed permission. An executed license
-- agreement with the author is required for all other uses of
-- this software. Redistribution of this software is not
-- permitted without the author's expressed permission.
-- This copyright notice must remain intact.
-- Derivative works may contain additional notices.
--
-- This software comes with no warranty.
--
--* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
-- bs_reg.vhd
--
-- Contains the code for the bs register block used to capture BS
-- signal from M32R for the state machine.
--
--*****************************************************************
--

library ieee;
use ieee.std_logic_1164.all;
library work;
```

```
use work.comp_pack.all;

entity bs_reg is
port(
  m32bs : in std_logic;

--BS delayed by one clock
  bs_d1 : out std_logic;

--BS delayed by one or more clocks for refresh condition

  bs_dm : out std_logic;

  clk : in std_logic;
  reset : in std_logic
);
end entity;


architecture beh of bs_reg is

signal int_reg1_out,int_reg2_out,int_reg3_out : std_logic;

begin

--Used to capture bs for everything but refreshing

  bs_d1 <= int_reg1_out;

--Used to capture bs when refreshing

  bs_dm <= int_reg1_out and int_reg2_out and int_reg3_out;

--Set of three registers

  reg1 : process(clk)
  begin

    if reset = '0' then

      int_reg1_out <= '0';

    elsif falling_edge(clk) then


      int_reg1_out <= m32bs;


    end if;

  end process;


  reg2 : process(clk)
  begin

    if reset = '0' then

        int_reg2_out <= '0';

    elsif falling_edge(clk) then

      int_reg2_out <= int_reg1_out;

    end if;

  end process;


  reg3 : process(clk)
  begin

    if reset = '0' then

      int_reg3_out <= '0';

    elsif falling_edge(clk) then

      int_reg3_out <= int_reg2_out;

    end if;

  end process;


end architecture;
```

E.7 Data Path Module

```
--****************************************************************
```

```
--CRAM Controller
--
-- Copyright (c) 2001 by Noah Aklilu, Edmonton, AB, Canada
-- All rights reserved.
-- This software may be used for non-profit university research
-- if given the author's expressed permission.  An executed license
-- agreement with the author is required for all other uses of
-- this software.  Redistribution of this software is not
-- permitted without the author's expressed permission.
-- This copyright notice must remain intact.
-- Derivative works may contain additional notices.
--
-- This software comes with no warranty.
--
--* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
-- databus_path.vhd
--
-- Contains the path for transferring data between the m32r
-- databus and the CRAM databus
--
--*****************************************************************
--

library ieee;
use ieee.std_logic_1164.all;

entity databus_path is
port (
  m32db_in : in std_logic_vector(0 to 15);
  m32db_out : out std_logic_vector(0 to 15);
  enable_m32db : in std_logic;
  load_msb : in std_logic;
  ppu_dram_transfer : in std_logic;
  ppu_ctrl_reg : in std_logic;
  clk : in std_logic;
  reset : in std_logic;
  cramdb_in : in std_logic_vector(31 downto 0);
  cramdb_out : out std_logic_vector(31 downto 0);
  enable_cramdb : in std_logic;
  data_word_sel : in std_logic
);
end entity;

architecture beh of databus_path is

signal int_msb_reg : std_logic_vector(0 to 15);
signal int_cramdb_out : std_logic_vector(31 downto 0);
signal int_m32db_out : std_logic_vector(0 to 15);

begin

---------------------
-- From M32r to CRAM
---------------------

int_cramdb_out(31 downto 16) <= int_msb_reg;

-- Mask unused bits
  masking : process(ppu_dram_transfer, ppu_ctrl_reg) is
  begin

-- Fix operations to PPU<->DRAM and never SOR<->DRAM

    if(ppu_dram_transfer = '1') then

      int_cramdb_out(15 downto 0) <= m32db_in;
      int_cramdb_out(6) <= '0';

-- Fix bits to prevent chunky mode access

    elsif(ppu_ctrl_reg = '1') then

      int_cramdb_out(15 downto 0) <= m32db_in;
      int_cramdb_out(1) <= '0';
      int_cramdb_out(7) <= '0';

-- Other options pass data through

    else

      int_cramdb_out(15 downto 0) <= m32db_in;

    end if;

  end process;

--MSB Reg

  msb_reg : process (clk) is

  begin
```

```
    if reset = '0' then

      int_msb_reg <= (others => '0');

    elsif falling_edge(clk) then

      if load_msb = '1' then

        int_msb_reg <= m32db_in;

      end if;

    end if;

  end process;

--CRAM Output Tristate Buffer

  with enable_cramdb select
    cramdb_out <= int_cramdb_out when '1',
            (others => 'Z') when others;


-----------------------------
--CRAM to M32R
-----------------------------

--Mux to select word portion of cram bus to pass

  with data_word_sel select
    int_m32db_out <= cramdb_in(31 downto 16) when '0',
             cramdb_in(15 downto 0) when others;

--M32R Output Tristate Buffer

  with enable_m32db select
    m32db_out <= int_m32db_out when '1',
          (others => 'Z') when others;


end architecture;
```

E.8 Refresh Monitor Module

```
--***************************************************************
--CRAM Controller
--
-- Copyright (c) 2001 by Noah Aklilu, Edmonton, AB, Canada
-- All rights reserved.
-- This software may be used for non-profit university research
-- if given the author's expressed permission. An executed license
-- agreement with the author is required for all other uses of
-- this software. Redistribution of this software is not
-- permitted without the author's expressed permission.
-- This copyright notice must remain intact.
-- Derivative works may contain additional notices.
--
-- This software comes with no warranty.
--
--* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
-- refresh_mon.vhd
--
-- Contains the code for keeping track of refresh intervals and tracking
-- addresses
--
-- Note: The timer keeps running even after a timeout, this is to prevent a
-- drift inserted by the timer waiting for the state machine to respond
--***************************************************************
--

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
library work;
use work.comp_pack.all;

entity refresh_mon is
port (
  clk : in std_logic;
  reset : in std_logic;

-- Increment Refresh Address signal
  inc_ref_add : in std_logic;

-- Current Refresh Address
  ref_add : out std_logic_vector(9 downto 0);

-- Refresh Timeout signal
  ref_timeout : out std_logic
);
```

```
end entity;

architecture beh of refresh_mon is

-- Used by the address counter
signal int_ref_add : std_logic_vector(9 downto 0);
constant end_address : std_logic_vector(9 downto 0) := "1100101111";

-- Used by the refresh interval counter
signal int_ref_timeout : std_logic;

-- Used by the refresh interval counter : Set for 250 clock cycles = 15us
-- Modify as needed

signal int_ref_count : std_logic_vector(7 downto 0);
constant end_value : std_logic_vector(7 downto 0) := "11111010";

begin

  ref_add <= int_ref_add;
  ref_timeout <= int_ref_timeout;

-------------------------------------------------
-- Refresh Time Interval counter
-------------------------------------------------

  ref_timer : process (clk)
  begin

-- clear things

    if reset = '0' then

      int_ref_timeout <= '0';
      int_ref_count <= (others => '0');

    elsif falling_edge(clk) then

-- Normal Interval Counting

        if (int_ref_count < end_value) then

          int_ref_count <= int_ref_count + 1;

-- Previous timeout has been acknowledged by state machine

          if (inc_ref_add = '1') and (int_ref_timeout = '1') then

            int_ref_timeout <= '0';

          end if;

-- Time out has occurred

        else

          int_ref_count <= (others => '0');

          int_ref_timeout <= '1';

        end if;

    end if;

  end process;


--------------------------------------------------------------
--  Refresh Address Counter
--------------------------------------------------------------

  ref_add_cnt : process (clk)
  begin

-- Clear things
    if reset = '0' then

      int_ref_add <= (others => '0');

    elsif falling_edge(clk) then

-- Signal to increment refresh address

      if (inc_ref_add = '1') then

        if (int_ref_add = end_address) then

          int_ref_add <= (others => '0');

        else
```

```
          int_ref_add <= int_ref_add + 1;

        end if;

      end if;

    end if;

  end process;

end architecture;
```

E.9 Control Logic Module

```
--***************************************************************
--CRAM Controller
--
-- Copyright (c) 2001 by Noah Aklilu, Edmonton, AB, Canada
-- All rights reserved.
-- This software may be used for non-profit university research
-- if given the author's expressed permission.  An executed license
-- agreement with the author is required for all other uses of
-- this software.  Redistribution of this software is not
-- permitted without the author's expressed permission.
-- This copyright notice must remain intact.
-- Derivative works may contain additional notices.
--
-- This software comes with no warranty.
--
--* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
-- statemachine.vhd
--
-- Contains the control logic (including the state machine) of the
-- controller
--
--***************************************************************
--

library ieee;
use ieee.std_logic_1164.all;
library work;
use work.comp_pack.all;

entity statemachine is
port (
-- Delayed BS signals
  m32bs_dl : in std_logic;
  m32bs_dm : in std_logic;

-- Address Decoder Signals
  read32 : in std_logic;
  write32 : in std_logic;
  read16 : in std_logic;
  write16 : in std_logic;

-- Refresh Timeout
  ref_timeout : in std_logic;

--M32R burst signal for 32 bit reads and writes
  m32burst : in std_logic;

-- Self-evident
  clk,reset : in std_logic;

--M32R Data Complete signal
  m32dc : out std_logic;

-- Enable CRAM DB output buffer
  enable_cramdb : out std_logic;

-- Enable M32R DB output buffer
  enable_m32db : out std_logic;

-- Signal to load MSB during 32bit writes
  load_msb : out std_logic;

--CRAM RAS and CAS signals
  cram_ras,cram_cas : out std_logic;

--CRAM Write Enable Signal
  cram_we : out std_logic;

-- Switch between Row and Col addresses
  cram_add_select : out std_logic;

-- Switch between MSB and LSB portions of CRAM word for M32R bus
  m32db_select : out std_logic;
-- Switch Refresh Address and Normal addressing
  cram_ref_select : out std_logic;

-- Increment Refresh Address counter
  inc_ref_add : out std_logic;

-- Debugging Signals
  led_out : out std_logic_vector(3 downto 0)

);
end entity;

architecture beh of statemachine is

type statevariable is (
-- Default start state
start,

-- 32 bit Write states
s1_32w, s2_32w,

-- 32 bit Read states
s1_32r, s2_32r, s2wait_32r, s3_32r, s4_32r,

-- 16 bit Read states
s1_16r, s2_16r, s2wait_16r, s3_16r,

-- 16 bit Write states
s1_16w, s2_16w,

-- Refresh states
s1_refresh, s2_refresh
);

signal state : statevariable;

signal int_led_out : std_logic_vector( 3 downto 0);

begin


  led_out <= int_led_out;

  main : process(clk) is
  begin

--
-- Using an async reset with altera labels a default start up state
--

    if reset = '0' then

      state <= start;

    elsif rising_edge(clk) then

--        if reset = '0' then

--          state <= start;

--        else

        case state is

        -- Detect conditions
          when start =>

          -- Refresh

            if ref_timeout = '1' then

              state <= s1_refresh;

          -- 16-bit Writes used for 8-bit data

            elsif (m32bs_d1 = '0') and (write16 = '1') and (m32burst = '1') then

              state <= s1_16w;

          -- 16-bit Read used for reading 8-bit data from Configuration Registers

            elsif (m32bs_d1 = '0') and (read16 = '1') and (m32burst ='1') then

              state <= s1_16r;

          -- 32-bit Reads

            elsif (m32bs_d1 = '0') and (read32 = '1') and (m32burst = '0') then

              state <= s1_32r;

          -- 32-bit Writes
        elsif (m32bs_d1 = '0') and (write32 = '1') and (m32burst = '0') then
          state <= s1_32w;
      -- Nothing Triggered
        else
          state <= start;
        end if;
    -- States for 32-bit Writes
      when s1_32w =>
        state <= s2_32w;
      when s2_32w =>
        state <= start;
    -- States for 32-bit Reads
      when s1_32r =>
        state <= s2_32r;
      when s2_32r =>
        state <= s2wait_32r;
      when s2wait_32r =>
        state <= s3_32r;
      when s3_32r =>
        state <= s4_32r;
      when s4_32r =>
        state <= start;
    -- States for 16-bit Reads
      when s1_16r =>
        state <= s2_16r;
      when s2_16r =>
        state <= s2wait_16r;
      when s2wait_16r =>
        state <= s3_16r;
      when s3_16r =>
        state <= start;
    -- States for 16-bit Writes
      when s1_16w =>
        state <= s2_16w;
      when s2_16w =>
        state <= start;

    -- States for handling refresh
      when s1_refresh =>
        state <= s2_refresh;
      when s2_refresh =>
      -- 16-bit Writes used for 8-bit data
        if (m32bs_dm = '0') and (write16 = '1') and (m32burst = '1') then
          state <= s1_16w;
      -- 16-bit Reads used for reading configuration registers
        elsif (m32bs_dm = '0') and (read16 = '1') and (m32burst = '1') then
```

```
                state <= s1_16r;

            -- 32-bit Reads

              elsif (m32bs_dm = '0') and (read32 = '1') and (m32burst = '0') then

                state <= s1_32r;

            -- 32-bit Writes

              elsif (m32bs_dm = '0') and (write32 = '1') and (m32burst = '0') then

                state <= s1_32w;

            -- Nothing Triggered

              else

                state <= start;

              end if;

            when others =>

              state <= start;

          end case;
--
-- "endif" associated with synchronous reset version
--        end if;

      end if;

  end process;

-- M32R/D Data Complete (DC) signal

  with state select
    m32dc <= '0' when s1_32w | s2_32w | s3_32r | s4_32r | s2_16w | s3_16r,
          '1' when others;

-- Enable the CRAM Databus output buffer

  with state select
    enable_cramdb <= '1' when s2_32w | s2_16w,
              '0' when others;

-- Send Write Enable signal to cram

  with state select
    cram_we <= '0' when s2_32w | s2_16w,
            '1' when others;

-- Enable the M32R Databus output buffer

  with state select
    enable_m32db <= '1' when s3_32r | s4_32r | s3_16r,
             '0' when others;

-- Load the MSB from M32R during 32-bit writes

  with state select
    load_msb <= '1' when s1_32w,
            '0' when others;

--CRAM RAS

  with state select
    cram_ras <= '0' when s1_32w | s2_32w | s1_32r | s2_32r | s2wait_32r
               | s3_32r | s4_32r | s1_16w | s2_16w | s1_refresh
               | s1_16r | s2_16r | s2wait_16r | s3_16r,
            '1' when others;
--CRAM CAS

  with state select
    cram_cas <= '0' when s2_32w | s2_32r | s2wait_32r | s3_32r | s4_32r | s2_16w
               | s2_16r | s2wait_16r | s3_16r,
            '1' when others;

-- Switch between row and col address for CRAM

  with state select
    cram_add_select <= '1' when s2_32w | s2_32r | s2wait_32r | s3_32r | s4_32r | s2_16w
                 | s2_16r | s2wait_16r | s3_16r,
               '0' when others;

-- Switch between M32R MSB and LSB portions of CRAM word

  with state select
    m32db_select <= '1' when s4_32r | s3_16r,
             '0' when others;
```

```
-- Switch between Refresh Address and Normal Addressing

  with state select
    cram_ref_select <= '1' when s1_refresh,
            '0' when others;

-- Increment refresh address

  with state select
    inc_ref_add <= '1' when s2_refresh,
             '0' when others;


-- Used to send debugging signals

  debugger : process (clk)
  begin

    if reset = '0' then

      int_led_out <= (others => '0');

    elsif falling_edge(clk) then

      if state = s2_32w then

        int_led_out <= ( 0 => '1', others => '0');

      elsif state = s4_32r then

        int_led_out <= ( 1 => '1', others => '0');

      elsif state = s2_16w then

        int_led_out <= ( 2 => '1', others => '0');

      elsif state = s3_16r then

        int_led_out <= ( 3 => '1', others => '0');

      else

        int_led_out <= int_led_out;

      end if;

    end if;

  end process;

end architecture;
```

E.10 Translation Block

```
--*****************************************************************
--CRAM Controller
--
-- Copyright (c) 2001 by Noah Aklilu, Edmonton, AB, Canada
-- All rights reserved.
-- This software may be used for non-profit university research
-- if given the author's expressed permission. An executed license
-- agreement with the author is required for all other uses of
-- this software. Redistribution of this software is not
-- permitted without the author's expressed permission.
-- This copyright notice must remain intact.
-- Derivative works may contain additional notices.
--
-- This software comes with no warranty.
--
--*************************************
-- translation_block.vhd
--
-- Contains the translation code for converting microprocessor
-- addresses into the corresponding address for the column and
-- row of the AX256
--
--*****************************************************************
--

library ieee;
use ieee.std_logic_1164.all;
library work;
use work.comp_pack.all;

entity translation_block is
port (
-- Microprocessor address bus
  add_bus : in std_logic_vector(13 to 30);

--Row and Column address outputs
```

```
  row_add : out std_logic_vector(9 downto 0);
  col_add : out std_logic_vector(9 downto 0)
);
end entity;

architecture beh of translation_block is

-- Microprocessor addresses

constant dram_low : std_logic_vector(9 downto 0) := "0000000000";
constant dram_upp : std_logic_vector(9 downto 0) := "1100101111";
constant src_reg : std_logic_vector(9 downto 0) := "1100110000";
constant dest_reg : std_logic_vector(9 downto 0) := "1100110001";
constant brush_reg : std_logic_vector(9 downto 0) := "1100110010";
constant mask_reg : std_logic_vector(9 downto 0) := "1100110011";
constant rop_reg : std_logic_vector(9 downto 0) := "1100110100";
constant fs_ppu_reg_low : std_logic_vector(17 downto 0) := "110011010100000000";
constant fs_ppu_reg_upp : std_logic_vector(17 downto 0) := "110011010100011110";
constant ppu_ctrl_reg : std_logic_vector(17 downto 0) := "110011010100100000";
constant banken_reg : std_logic_vector(17 downto 0) := "110011010100100001";
constant rastop_reg : std_logic_vector(17 downto 0) := "110011010100100010";
constant ropfs : std_logic_vector(17 downto 0) := "110011010100100011";
constant masksys : std_logic_vector(16 downto 0) := "11001101010010010";
constant ppu_tfr_reg : std_logic_vector(17 downto 0) := "110011010100100110";
constant fs_ctrl_reg : std_logic_vector(17 downto 0) := "110011010100100111";
constant fs_access_reg_low : std_logic_vector(16 downto 0) := "11001101010010101";
constant fs_access_reg_upp : std_logic_vector(16 downto 0) := "11001101010010111";

-- Real column addresses

constant real_src_reg : std_logic_vector(9 downto 0) := "1111000000"; -- 0x3C0
constant real_dest_reg : std_logic_vector(9 downto 0) := "1111000001"; -- 0x3C1
constant real_brush_reg : std_logic_vector(9 downto 0) := "1111000010"; -- 0x3C2
constant real_mask_reg : std_logic_vector(9 downto 0) := "1111000011"; -- 0x3C3
constant real_rop_reg : std_logic_vector(9 downto 0) := "1111000100"; -- 0x3C4
constant real_fs_ppu_reg : std_logic_vector(9 downto 0) := "1111000111"; -- 0x3C7
constant real_ppu_ctrl_reg : std_logic_vector(9 downto 0) := "1111001001"; -- 0x3C9
constant real_banken_reg : std_logic_vector(9 downto 0) := "1111001010"; -- 0x3CA
constant real_rastop_reg : std_logic_vector(9 downto 0) := "1111001100"; -- 0x3CC
constant real_ropfs : std_logic_vector(9 downto 0) := "1111001110"; -- 0x3CE
constant real_masksys : std_logic_vector(9 downto 0) := "1111001101"; -- 0x3CD
constant real_ppu_tfr_reg : std_logic_vector(9 downto 0) := "1111001111"; -- 0x3CF
constant real_fs_ctrl_reg : std_logic_vector(9 downto 0) := "1111000101"; -- 0x3C5
constant real_fs_access_reg : std_logic_vector(9 downto 0) := "1111000110"; -- 0x3C6


begin

-- Column address can be taken directly from the bits of the address space

  col_add(9 downto 3) <= add_bus(23 to 29);
  col_add(2 downto 0) <= "000";

-- Process for handling translation

  translation : process(add_bus)
  begin

-- DRAM storage

    if (add_bus(13 to 22) >= dram_low) and (add_bus(13 to 22) <= dram_upp) then

      row_add <= add_bus(13 to 22);


-- SRC PPU Register

    elsif (add_bus(13 to 22) = src_reg) then

      row_add <= real_src_reg;

-- DEST PPU Register

    elsif (add_bus(13 to 22) = dest_reg) then

      row_add <= real_dest_reg;

-- BRUSH PPU Register

    elsif (add_bus(13 to 22) = brush_reg) then

      row_add <= real_brush_reg;

-- MASK PPU Register

    elsif (add_bus(13 to 22) = mask_reg) then

      row_add <= real_mask_reg;

-- ROP PPU Register

    elsif (add_bus(13 to 22) = rop_reg) then
```

```
        row_add <= real_rop_reg;
-- FS <-> PPU Register
      elsif (add_bus(13 to 30) >=  fs_ppu_reg_low) and (add_bus(13 to 30) <= fs_ppu_reg_upp) then
        row_add <= real_fs_ppu_reg;
-- PPU Ctrl Register
      elsif (add_bus(13 to 30) = ppu_ctrl_reg) then
        row_add <= real_ppu_ctrl_reg;
-- BANKEN Register
      elsif (add_bus(13 to 30) = banken_reg) then
        row_add <= real_banken_reg;
-- RASTOP Register
      elsif (add_bus(13 to 30) = rastop_reg) then
        row_add <= real_rastop_reg;
-- ROPFS Register
      elsif (add_bus(13 to 30) = ropfs) then
        row_add <= real_ropfs;
-- MASKSYS Register
      elsif (add_bus(13 to 29) = masksys) then
        row_add <= real_masksys;
-- PPU Transfer Register
      elsif (add_bus(13 to 30) = ppu_tfr_reg) then
        row_add <= real_ppu_tfr_reg;
-- Funnel Shift Ctrl Register
      elsif (add_bus(13 to 30) = fs_ctrl_reg) then
        row_add <= real_fs_ctrl_reg;
-- Funnel Shift Access Register
      elsif (add_bus(13 to 29) >= fs_access_reg_low) and (add_bus(13 to 29) <= fs_access_reg_upp) then
        row_add <= real_fs_access_reg;
-- All other things
      else
        row_add <= add_bus(13 to 22);
      end if;
   end process;
end architecture;
```

Appendix F

Source Code for Kernels

F.1 Vector Addition

F.1.1 Ultra 10

```
/*****************************************************************
 * ESM Benchmarks
 *
 * Copyright (c) 2001 by Noah Aklilu, Edmonton, AB, Canada
 * All rights reserved.
 * This software may be used for non-profit university research
 * if given the author's expressed permission.  An executed license
 * agreement with the author is required for all other uses of
 * this software.  Redistribution of this software is not
 * permitted without the author's expressed permission.
 * This copyright notice must remain intact.
 * Derivative works may contain additional notices.
 *
 * This software comes with no warranty.
 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 * Workstation (Ultra10) version of vector addition benchmark
 *
 *****************************************************************
 */

#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>

/* Define size of arrays */

#define NUM_PE 4096
#define NUM_DATA 25

/* Static allocation of arrays for optimal data placement */

unsigned int a[NUM_DATA][NUM_PE], x[NUM_DATA][NUM_PE], b[NUM_DATA][NUM_PE];

int main(void)
{
  int i,j,count;
  long dtime;
  struct rusage time1,time2;

  /* Initialize arrays */

  for(i=0;i<NUM_DATA;i++)
    {
      for(j=0;j<NUM_PE;j++)
  {
    a[i][j] = x[i][j] = i*j;
  }

    }

  /* Resource usage snapshot */

  getrusage(RUSAGE_SELF,&time1);

  /* Perform operation */
  /* Repeat multiple times to minimize effect
     of measurement granularity */

  for(count=0;count<200;count++)
    {
      for(i=0;i<NUM_DATA;i++)
  {
    for(j=0;j<NUM_PE;j++)
      {
        b[i][j] = a[i][j] + x[i][j];
      }
  }
    }

  /* Resource usage snapshot */

  getrusage(RUSAGE_SELF,&time2);

  /* Time difference */

  dtime = ((time2.ru_utime.tv_sec * 1000000)+ time2.ru_utime.tv_usec) -
    ((time1.ru_utime.tv_sec * 1000000) + time1.ru_utime.tv_usec);

  /* Print results */

  printf("Total_Amount_of_time:_%ld_us\n", dtime);

  printf("Amount_of_time_per_cycle:_%ld_us\n",dtime/200);

  return 0;
}
```

F.1.2 MSA2000

```
/*****************************************************************
 * ESM Benchmarks
 *
 * Copyright (c) 2001 by Noah Aklilu, Edmonton, AB, Canada
 * All rights reserved.
 * This software may be used for non-profit university research
 * if given the author's expressed permission. An executed license
 * agreement with the author is required for all other uses of
 * this software. Redistribution of this software is not
 * permitted without the author's expressed permission.
 * This copyright notice must remain intact.
 * Derivative works may contain additional notices.
 *
 * This software comes with no warranty.
 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 * M32R version of the vector addition benchmark
 *
 *******************************************************************
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "timer.h"

/* Define array size */

#define NUM_PE 4096
#define NUM_DATA 25

/* Static allocation of arrays */

unsigned int a[NUM_DATA][NUM_PE],x[NUM_DATA][NUM_PE],b[NUM_DATA][NUM_PE];

int main(void)
{
  struct timeval value;
  int i,j,k;

  printf("Starting_up\n");

  /* Initialize arrays */

  for(i=0;i<NUM_DATA;i++)
    {
      for(j=0;j<NUM_PE;j++)
  {
    a[i][j] = x[i][j] = i*j;
  }

    }

  /* Start Timer */

  timerStart();
```

```
    for(i=0;i<NUM_DATA;i++)
      {
        for(j=0;j<NUM_PE;j++)
    {
      b[i][j] = a[i][j] + x[i][j];
    }
      }


    /* Stop Timer */

    timerStop();

    /* Retrieve Timer Value */

    timerTime(&value);

    /* Print results */

    printf("Time:_%d_secs_%d_usecs\n", value.tv_sec, value.tv_usec);

    timerClear();

    return 0;

}
```

F.1.3 Embedded SIMD Machine

```
/****************************************************************
 * ESM Benchmarks
 *
 * Copyright (c) 2001 by Noah Aklilu, Edmonton, AB, Canada
 * All rights reserved.
 * This software may be used for non-profit university research
 * if given the author's expressed permission.  An executed license
 * agreement with the author is required for all other uses of
 * this software.  Redistribution of this software is not
 * permitted without the author's expressed permission.
 * This copyright notice must remain intact.
 * Derivative works may contain additional notices.
 *
 * This software comes with no warranty.
 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 * March 7, 2001
 * Noah Aklilu
 *
 * CRAM version of the vector addition benchmark
 *
 ******************************************************************
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "cram_mem.h"
#include "timer.h"

int main(void)
{
  volatile unsigned int *a;
  volatile unsigned int *x;
  volatile unsigned int *b;
  int i,j;
  struct timeval t_value;

  a = STARTDRAM;
  x = a + 16;
  b = a + 32;


  /* Measure Array Addition */

  printf("Beginning_Array_Addition\n");

  /* Start and Setup Timer */

  timerClear();

  timerStart();

  /* Enable all Banks */

  *BANKENREG = 0xFFFF;

  /* Enable all PE words */

  *PPUCTRLREG = 0x0004;
```

```
    /* Write a one to MASK register */

    *MASKREG = 0xFFFFFFFF;

    /* Begin High level loop */

    for(i=0;i<25;i++)
      {

        /* Write a "0" to carry register */

        *DESTREG =  0x00000000;

        /* Begin Bit Level loop */

        for(j=0;j<32;j++)
    {
      /* Load a register */

      *a = 0x00000084;

      /* Load x register */

      *x = 0x00000081;

      /* Load sum opcode */

      *RASTOPREG = 0x0096;

      /* Perform operation */

      *ROPFS = 0x0000;

      /* save result register */

      *b = 0x00000010;

      /* load carry opcode */

      *RASTOPREG = 0x00E8;

      /* perform operation */

      *ROPFS = 0x0000;

      /* move result to carry register */

      *PPUTFRREG = 0x0082;

      /* Increment addresses */

      a = a + 128;
      x = x + 128;
      b = b + 128;

    }

      }

    /* Stop Timer */

    timerStop();

    printf("End_Array_operation\n");

    timerTime(&t_value);

    printf("_Vector_Addition_Time:_%d_secs_%d_usecs\n",
     t_value.tv_sec,t_value.tv_usec);

    timerClear();

    return 0;

}
```

F.2 Parallel Multiplication

F.2.1 Ultra 10

```
/***************************************************************
  * ESM Benchmarks
  *
  * Copyright (c) 2001 by Noah Aklilu, Edmonton, AB, Canada
  * All rights reserved.
  * This software may be used for non-profit university research
  * if given the author's expressed permission. An executed license
```

```
 * agreement with the author is required for all other uses of
 * this software.  Redistribution of this software is not
 * permitted without the author's expressed permission.
 * This copyright notice must remain intact.
 * Derivative works may contain additional notices.
 *
 * This software comes with no warranty.
 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 * Workstation (Ultra10) version of parallel multiplication benchmark
 *
 *****************************************************************
 */

#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>

/* Size of array */

#define NUM_PE 4096
#define NUM_DATA 25

/* Static allocatioin to take advantage of optimial data placement */

unsigned int a[NUM_DATA][NUM_PE], x[NUM_DATA][NUM_PE], b[NUM_DATA][NUM_PE];

int main(void)
{
  int i,j,count;
  long dtime;
  struct rusage time1,time2;

  /* Fill arrays with data */

  for(i=0;i<NUM_DATA;i++)
    {
      for(j=0;j<NUM_PE;j++)
  {
    a[i][j] = x[i][j] = i*j;
  }

    }

  /* Take resource usage snapshot before starting */

  getrusage(RUSAGE_SELF,&time1);

  /* Perform operation */
  /* Repeat multiple times to reduce effect of time granularity */

  for(count=0;count<200;count++)
    {
      for(i=0;i<NUM_DATA;i++)
  {
    for(j=0;j<NUM_PE;j++)
      {
        b[i][j] = a[i][j] * x[i][j];
      }
  }
    }

  /* Snapshot aftewards */

  getrusage(RUSAGE_SELF,&time2);

  /* Calculate time difference between snapshots */

  dtime = ((time2.ru_utime.tv_sec * 1000000)+ time2.ru_utime.tv_usec) -
    ((time1.ru_utime.tv_sec * 1000000) + time1.ru_utime.tv_usec);

  /* Print results */

  printf("Total Amount of time: %ld us\n", dtime);

  printf("Amount of time per cycle: %ld us\n",dtime/200);

  return 0;
}
```

F.2.2 MSA2000

```
/*****************************************************************
 * ESM Benchmarks
 *
 * Copyright (c) 2001 by Noah Aklilu, Edmonton, AB, Canada
 * All rights reserved.
 * This software may be used for non-profit university research
 * if given the author's expressed permission.  An executed license
 * agreement with the author is required for all other uses of
 * this software.  Redistribution of this software is not
 * permitted without the author's expressed permission.
 * This copyright notice must remain intact.
 * Derivative works may contain additional notices.
 *
 * This software comes with no warranty.
 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 * M32R version of the parallel multiplication benchmark
 *
 *********************************************************************
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "timer.h"

/* Define the size of the array */

#define NUM_PE 4096
#define NUM_DATA 25

/* Static allocation of arrays */

unsigned int a[NUM_DATA][NUM_PE], x[NUM_DATA][NUM_PE], b[NUM_DATA][NUM_PE];

int main(void)
{
  struct timeval value;
  int i,j;

  printf("Starting up\n");

  /* Initialize arrays */

  for(i=0;i<NUM_DATA;i++)
    {
      for(j=0;j<NUM_PE;j++)
  {
    a[i][j] = x[i][j] = i*j;
  }

    }

  /* Start Timer */

  timerStart();

  /* Perform operation */

  for(i=0;i<NUM_DATA;i++)
    {
      for(j=0;j<NUM_PE;j++)
  {
    b[i][j] = a[i][j] * x[i][j];
  }
    }

  /* Stop timer */

  timerStop();

  /* Retrieve Timer value */

  timerTime(&value);

  /* Print out results */

  printf("Time: %d secs %d usecs\n", value.tv_sec, value.tv_usec);

  timerClear();

  return 0;

}
```

F.2.3 Embedded SIMD Machine

```
/*****************************************************************
 * ESM Benchmarks
 *
 * Copyright (c) 2001 by Noah Aklilu, Edmonton, AB, Canada
 * All rights reserved.
 * This software may be used for non-profit university research
 * if given the author's expressed permission.  An executed license
 * agreement with the author is required for all other uses of
 * this software.  Redistribution of this software is not
 * permitted without the author's expressed permission.
 * This copyright notice must remain intact.
 * Derivative works may contain additional notices.
 *
 * This software comes with no warranty.
 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 * CRAM version of the parallel multiplication benchmark
 *
 *********************************************************************
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "cram_mem.h"
#include "timer.h"

int main(void)
{
  volatile unsigned int *a;
  volatile unsigned int *x;
  volatile unsigned int *b;
  volatile unsigned int *a_summer;
  volatile unsigned int *b_summer;
  volatile unsigned int *int_b_summer;
  struct timeval t_value;
  int i,j,k;

  a = STARTDRAM;
  x = a + 16;
  b = a + 32;


  /* Measure Parallel Multiplication */

  printf("Beginning_Parallel_Multiplication\n");

  /* Setup Timer */

  timerClear();

  /* Enable all Banks */

  *BANKENREG = 0xFFFF;

  /* Enable all PE words */

  *PPUCTRLREG = 0x0004;

  /* Clear B register */

  b_summer = b;

  for(i=0;i<816;i++)
    {
      *b_summer = 0x00000000;
      b_summer = b_summer + 128;
    }

  /* Start Timer */

  timerStart();

  /* Begin High level loop */

  for(i=0;i<25;i++)
    {

      b_summer = b;

      for(j=0;j<32;j++)
  {

    a_summer = a;
    int_b_summer = b_summer;

    /* Load x into the MASK register */

    *x = 0x00000088;

    x = x + 128;

    /* Write a "0" to carry register */

    *DESTREG =  0x00000000;

    /* Begin Bit Level loop */

    for(k=j;k<32;k++)
      {
            /* Load a_summer into BRUSH register */

            *a_summer = 0x00000084;

            /* Load b_summer into DEST regiser */

            *int_b_summer = 0x00000081;

            /* Load sum opcode */

            *RASTOPREG = 0x0096;

            /* Perform operation */

            *ROPFS = 0x0000;

            /* save result register */

            *int_b_summer = 0x00000002;

            /* load carry opcode */

            *RASTOPREG = 0x00E8;

            /* perform operation */

            *ROPFS = 0x0000;

            /* move carry result to SRC register */

            *PPUTFRREG = 0x0081;

            /* Increment addresses */

            a_summer = a_summer + 128;
            int_b_summer = int_b_summer + 128;

          }

        /* Increment start point in summation register */

        b_summer = b_summer + 128;

      }

          /* Increment variables */

          a = a + 4096;
          b = b + 4096;

        }

      /* Stop Timer */

      timerStop();

      printf("End_Array_operation\n");

      timerTime(&t_value);

      printf("_Parallel_Multiplication_Time:_%d_secs_%d_usecs\n",
       t_value.tv_sec,t_value.tv_usec);

      timerClear();

      return 0;

    }
```

F.3 Vector Multiplication

F.3.1 Ultra 10

```
/*****************************************************************
 * ESM Benchmarks
 *
 * Copyright (c) 2001 by Noah Aklilu, Edmonton, AB, Canada
 * All rights reserved.
 * This software may be used for non-profit university research
 * if given the author's expressed permission.  An executed license
 * agreement with the author is required for all other uses of
 * this software.  Redistribution of this software is not
 * permitted without the author's expressed permission.
 * This copyright notice must remain intact.
 * Derivative works may contain additional notices.
 *
 * This software comes with no warranty.
 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 * Workstation (Ultra10) version of vector multiplication benchmark
 *
 ********************************************************************
 */

#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include <sys/resource.h>

/* Define the size of the arrays */

#define NUM_PE 4096
#define NUM_DATA 25

/* Use static allocation of arrays for optimial data placement */

unsigned int a[NUM_DATA][NUM_PE], x[NUM_DATA], b[NUM_PE];

int main(void)
{
  int i,j,count;
  long dtime;
  struct rusage time1,time2;

  /* Initialize arrays */

  for(i=0;i<NUM_DATA;i++)
    {
      for(j=0;j<NUM_PE;j++)
  {
    a[i][j] = i*j;
    b[j] = 0;
  }

      x[i] = i*25;
    }

  /* Take resource usage snapshot */

  getrusage(RUSAGE_SELF,&time1);

  /* Perform operation */
  /* Repeat multiple times to reduce effect
     of measurement granularity */

  for(count=0;count<200;count++)
    {
      for(i=0;i<NUM_DATA;i++)
  {
    for(j=0;j<NUM_PE;j++)
      {
        b[j] =+ a[i][j]*x[i];
      }
  }
    }

  /* Take resource usage snapshot */

  getrusage(RUSAGE_SELF,&time2);

  /* Calculate time difference between snapshots */

  dtime = ((time2.ru_utime.tv_sec * 1000000)+ time2.ru_utime.tv_usec) -
    ((time1.ru_utime.tv_sec * 1000000) + time1.ru_utime.tv_usec);

  /* Print results */

  printf("Total_Amount_of_time:_%ld_us\n", dtime);

  printf("Amount_of_time_per_cycle:_%ld_us\n",dtime/200);

  return 0;
}
```

F.3.2 MSA2000

```
/*****************************************************************
 * ESM Benchmarks
 *
 * Copyright (c) 2001 by Noah Aklilu, Edmonton, AB, Canada
 * All rights reserved.
 * This software may be used for non-profit university research
 * if given the author's expressed permission.  An executed license
 * agreement with the author is required for all other uses of
 * this software.  Redistribution of this software is not
 * permitted without the author's expressed permission.
 * This copyright notice must remain intact.
 * Derivative works may contain additional notices.
 *
 * This software comes with no warranty.
 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 * M32R version of the vector multiplication benchmark
 *
 *******************************************************************
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "timer.h"

/* Define size of arrays */

#define NUM_PE 4096
#define NUM_DATA 25

/* Static Allocation of arrays */

unsigned int a[NUM_DATA][NUM_PE], x[NUM_DATA], b[NUM_PE];

int main(void)
{
  struct timeval value;
  int i,j;

  printf("Starting up\n");

  /* Initialize arrays */

  for(i=0;i<NUM_DATA;i++)
    {
      for(j=0;j<NUM_PE;j++)
  {
    a[i][j] = i*j;
    b[j] = 0;
  }

      x[i] = i*25;

    }

  /* Start Timer */

  timerStart();

  /* Perform operation */

  for(i=0;i<NUM_DATA;i++)
    {
      for(j=0;j<NUM_PE;j++)
  {
    b[j] =+ (a[i][j] * x[i]);
  }
    }

  /* Stop Timer */

  timerStop();

  /* Retrieve Timer value */

  timerTime(&value);

  /* Print results */

  printf("Time: %d secs %d usecs\n", value.tv_sec, value.tv_usec);

  timerClear();

  return 0;

}
```

F.3.3 Embedded SIMD Machine

```
/****************************************************************
 * ESM Benchmarks
 *
 * Copyright (c) 2001 by Noah Aklilu, Edmonton, AB, Canada
 * All rights reserved.
 * This software may be used for non-profit university research
 * if given the author's expressed permission. An executed license
 * agreement with the author is required for all other uses of
 * this software. Redistribution of this software is not
 * permitted without the author's expressed permission.
 * This copyright notice must remain intact.
 * Derivative works may contain additional notices.
 *
 * This software comes with no warranty.
 *
 * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 * CRAM version of the vector multiplication benchmark
 *
 ********************************************************************
 */


#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "cram_mem.h"
#include "timer.h"

int main(void)
{
  volatile unsigned int *a;
  volatile unsigned int *x;
  volatile unsigned int *b;
  volatile unsigned int *a_summer;
  volatile unsigned int *b_summer;
  volatile unsigned int *int_b_summer;
  struct timeval t_value;
  int i,j,k;

  a = STARTDRAM;
  x = a + 16;
  b = a + 32;


  /* Measure Parallel Multiplication */

  printf("Beginning_Parallel_Multiplication\n");

  /* Setup Timer */

  timerClear();


  /* Enable all Banks */

  *BANKENREG = 0xFFFF;

  /* Enable all PE words */

  *PPUCTRLREG = 0x0004;

  /* Clear B register */

  b_summer = b;

  for(i=0;i<816;i++)
    {
      *b_summer = 0x00000000;
      b_summer = b_summer + 128;
    }

  /* Start Timer */

  timerStart();

  /* Begin High level loop */

  for(i=0;i<25;i++)
    {

      b_summer = b;

      for(j=0;j<32;j++)
  {

    a_summer = a;
    int_b_summer = b_summer;

    /* Load x into the MASK register */

    *x = 0x00000088;

    x = x + 128;

    /* Write a "0" to carry register */

    *DESTREG =  0x00000000;

    /* Begin Bit Level loop */

    for(k=j;k<32;k++)
      {
            /* Load a_summer into BRUSH register */

            *a_summer = 0x00000084;

            /* Load b_summer into DEST regiser */

            *int_b_summer = 0x00000081;

            /* Load sum opcode */

            *RASTOPREG = 0x0096;

            /* Perform operation */

            *ROPFS = 0x0000;

            /* save result register */

            *int_b_summer = 0x00000002;

            /* load carry opcode */

            *RASTOPREG = 0x00E8;

            /* perform operation */

            *ROPFS = 0x0000;

            /* move carry result to SRC register */

            *PPUTFRREG = 0x0081;

            /* Increment addresses */

            a_summer = a_summer + 128;
            int_b_summer = int_b_summer + 128;

          }

        /* Increment start point in summation register */

        b_summer = b_summer + 128;

      }

          /* Increment variables */

          a = a + 4096;
          b = b + 4096;

        }

      /* Summation of the multiplication results */
      /* b = SUM(b_x) */

      /* Start at the first element of b */

      b_summer = STARTDRAM + 32;

      /* Write a one to MASK register */

      *MASKREG = 0xFFFFFFFF;

      /* Variable level loop */

      for(i=1;i<25;i++)
        {
          int_b_summer = STARTDRAM + 32;

          /* Write a "0" to carry register */

          *DESTREG = 0x00000000;

          /* Bit level loop */

          for(j=0;j<32;j++)
      {

        /* Load a register */

        *b_summer = 0x00000084;

        /* Load x register */

        *int_b_summer = 0x00000081;

        /* Load sum opcode */

        *RASTOPREG = 0x0096;

        /* Perform operation */
      *ROPFS = 0x0000;

      /* save result register */

      *int_b_summer = 0x00000010;

      /* load carry opcode */

      *RASTOPREG = 0x00E8;

      /* perform operation */

      *ROPFS = 0x0000;

      /* move result to carry register */

      *PPUTFRREG = 0x0082;

      /* Increment addresses */

      int_b_summer = int_b_summer + 128;
      b_summer = b_summer + 128;

    }
      }

    /* Stop Timer */

    timerStop();

    /* Print out results */

    printf("_End_Array_operation\n");

    timerTime(&t_value);

    printf("_Vector_Multiplication_Time:_%d_secs_%d_usecs\n",
     t_value.tv_sec, t_value.tv_usec);

    timerClear();

    return 0;

}
```